ARL-TR-14

AR-007-135

DEPARTMENT OF DEFENCE

DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION

AERONAUTICAL RESEARCH LABORATORY

MELBOURNE, VICTORIA

Technical Report 14

A SOFTWARE INTERFACE FOR THE ARL WIND TUNNEL DATA ACQUISITION SYSTEM

by

B.D. FAIRLIE
S.S.W. LAM

Approved for public release.

MARCH 1993

AR-007-135

DEPARTMENT OF DEFENCE
DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION
AERONAUTICAL RESEARCH LABORATORY

Technical Report 14

A SOFTWARE INTERFACE FOR THE ARL WIND TUNNEL DATA ACQUISTION SYSTEM

by

B.D. FAIRLIE
S.S.W. LAM

SUMMARY

A software interface for the data acquisition system has been developed on a MicroVAX II computer for the Transonic and Low Speed wind tunnels at ARL. The software is responsible for handling instrumentation control and data transfer requests between the data acquisition software and the parallel data bus via a DRV11 parallel I/O interface adapter. Access to the DRV11 registers is effected by direct mapping of the Q22-Bus I/O page to program variables, giving fast and efficient transfer of data to and from the parallel data bus. Up to five processes may access the parallel data bus at one time via this software interface thus allowing great flexibility in the development of data acquisition software. This report details the necessary programming steps which must be included in data acquisition software to access the parallel data bus via the software interface.

POSTAL ADDRESS: **Director, Aeronautical Research Laboratory,**
506 Lorimer Street, Fishermens Bend, 3207
Victoria, Australia.

CONTENTS

1 INTRODUCTION

The use of computers for data acquisition and analysis in wind tunnels at the Aeronautical Research Laboratory (ARL) began in the early 1960s. The early Transonic Wind Tunnel (VPT) instrumentation was built around a Digital Equipment Corporation (DEC[1]) PDP[1]-8/I computer and automated much of the testing procedure and data analysis. This system was in operation from 1967 until the late 1980s by which time the PDP-8/I computer had well and truly reached the end of its life. It was then replaced by a DEC MicroVAX[1]II micro-computer. Much of the instrumentation that was built around the PDP-8/I was also replaced.

In the Subsonic Wind Tunnel (LSWT), up until the 1960s analysis of data was carried out in an off-line batch operation on an external computer. In the early 1970s the first on-line data acquisition system was installed, which collected data from selected data sources in a pre-arranged sequence. The data was subsequently analysed off-line on the central site computer. Later, a serial line connecting the system to the central site computer allowed data to be processed in "real-time" and the results of the test displayed on a terminal screen almost immediately after the data had been collected. In 1982, a dedicated DEC PDP-11/44 mini-computer was installed for the sole purpose of wind tunnel data acquisition and processing, and tunnel control. Comprehensive software has been written and developed over the years to take advantage of the computing power available on this dedicated system. Up until the late 1980s data were, however, still being collected via a relatively slow and cumbersome "serializer" which had become outdated and unreliable.

To maintain the productivity and quality of the work that has been carried out in these wind tunnels, a new data acquisition system based on microprocessor technology was designed at ARL and installed in both tunnels. The new system comprises of several microprocessor based instrumentation modules (slave modules) connected to a dedicated master (host) computer via a bi-directional differential parallel data bus also developed at ARL [1]. Reference [1] describes this system and reasons for selecting a master slave system with the intelligence distributed across the bus. This report describes the software that provides an interface between the parallel data bus and the data acquisition programs which run on the master computer. The software is written in VAX[1]FORTRAN but the ideas and logic could readily be adopted to other high level computer languages.

2 THE HARDWARE INTERFACE

Communication between the MicroVAX II computer and the tunnel instrumentation modules is provided by a bi-directional differential parallel (BDP) bus system with appropriate interfaces as shown in Figure 1. The BDP data bus is a 16-bit wide bus that provides input, output, control and addressing functions. It operates in a master/slave format in which the master initiates all traffic on the address bus. It allows input and output to be sent to or received from any individual tunnel instrumentation module (slave), in any order, with the timing controlled by the MicroVAX II computer (master). The slave modules cannot address the master, which eliminates the possibility of bus hang-ups caused by two or more modules activating the bus simultaneously. However, the slaves can signal the master, requesting its attention

[1]DEC, PDP, MicroVAX and VAX are registered trademarks of Digital Equipment Corporation

via a common error/flag line. The master's attention is requested to warn of an error, such as invalid data, to indicate that data gathering has been completed or that a control function setting has been finalized.

The parallel input/output interface between the Q22-Bus[2] of the MicroVAX II computer and the wind tunnel data bus is provided by a DEC DRV11 General Device Interface Adapter. There are several variants of the DRV11 Adapter available on the MicroVAX II computer. The one used here is the Q22-Bus equivalent of the UNIBUS[2] DR11-C interface adapter, designated as DEC module number M7941. There is, however, a design fault in the circuit board and modifications must be made, which are detailed in Appendix A. The DRV11 is a general-purpose digital interface which provides 16 input and 16 output lines (for handling 16-bit parallel data), and 8 control lines (for transferring control and status information) to transistor-transistor logic (TTL) compatible external devices. These input/output and control lines are converted to a bi-directional differential parallel bus via a single self-contained card interface — the DRV11 Bus Interface card, also developed at ARL [1]. Through this hardware arrangement the MicroVAX II computer can communicate with the slave modules at a rate greater than 1800 16-bit data words per second [1].

3 A DRV11 "DEVICE DRIVER"

An interface adapter attached to the Q22-Bus of a MicroVAX II is usually accessed from a computer program via a device driver that is part of the VMS[2] operating system. However, the DRV11 is not supported by VMS, and hence a device driver is not available. A different approach must therefore be employed to effect the transfer of data and controls from user written programs to the DRV11 interface adapter.

The DRV11 has three registers for control and data transfer [2]:

- the control and status register (CSR)
- the output buffer
- the input buffer.

These registers may be accessed from within a computer program by mapping the MicroVAX Q22-Bus input/output (I/O) page[3], which contains the addresses of the device registers above, to variables appropriately defined in the program. Values assigned to these variables are written directly into the respective registers and data received by the input buffer may be read directly from the variable mapped to it. This technique of accessing the registers of a device directly is very efficient [3], because it by-passes the VMS operating system's optimization and resource management processes. However, the technique is also dangerous if not used correctly, as it also by-passes the normal checking and protection procedures of the operating system.

The two interrupt vectors of the DRV11 interface may be connected to a process[4] by a simple device handler, `CONINTERR`, through the VMS `SYSGEN` facility [4]. The

[2]Q22-Bus, UNIBUS and VMS are registered trademarks of Digital Equipment Corporation

[3]A page is a set of 512 contiguous byte locations used as the unit of memory mapping and protection.

[4]VMS refers to any task under the control of the operating system, including any routines that may be executed as part of the running a scheduled job as a *process*.

parallel data bus uses one interrupt to indicate to the master that *data are ready*, and the other to notify the master of any *error* detected in the instrumentation.

3.1 Mapping The Input/Output Page

To gain access to the I/O page (or for any page of physical memory) a process must map the page into its virtual address space. In the present case, the page of interest is that containing the DRV11 registers. The VMS operating system provides a system service routine SYS$CRMPSC (create and map section) to perform the mapping of I/O space to a process's memory. This routine is called as follows:

```
 status = SYS$CRMPSC([inadr],[retads],[acmode],[flags],[gsnam],
1          [ident],[relpag],[chan],[pagecnt],[vbn],[prot],[pfc])
```

where status is an INTEGER*4 variable returned by the routine to indicate the status of the operation. According to the convention used in the *MicroVMS Programmer's Manual* [5], items enclosed in square brackets are optional. The parameters relpag, chan and pfc are not relevant to mapping the I/O space and can be left blank. The other arguments have the following meaning:

inadr — An array of two longwords[5] containing the addresses of the beginning and end of the region in virtual memory to which the I/O page is to be mapped. Thus, to map a single page of I/O space to an array variable, iopage, the following code fragment may be used.

```
INTEGER*2 iopage(0:255)
INTEGER*4 inadr(2)

inadr(1) = %LOC ( iopage(0) )
inadr(2) = %LOC ( iopage(255) ) + 1
```

The variable iopage to which the I/O page will be mapped is 256 words long since there are 512 bytes in each page. inadr(1) is the address of the beginning of the region and inadr(2) is set to the address of iopage(255) plus one, so that it contains the last byte of the region.

retadr — An array of two longwords which on return from the routine SYS$CRMSPC will contain the addresses of the beginning and the end of the region of virtual memory that the I/O page was actually mapped to. Note that if retadr(1) $\neq$ inadr(1) or retadr(2) $\neq$ inadr(2), then an error has occurred, usually due to mis-specification of one or more parameters. If the mapping has not been done at all, both values in retadr are returned as -1.

acmode — This parameter specifies the access mode to be applied to the mapped region. It is optional and is omitted in the present case. The access mode then defaults to that of the calling process.

flags — A longword containing a mask which determines the characteristics of the mapped section. When mapping the I/O page, flags should be set up with the following code fragment

[5] A longword is a data unit corresponding to 4 bytes of memory, usually declared as an INTEGER*4 variable in VAX FORTRAN.

```
INCLUDE '($SECDEF)'
flags = SEC$M_PFNMAP .OR. SEC$M_WRT
```

The first statement includes the system definition library module containing the definitions of the bit offsets used in the statement following to set the variable `flags`. `SEC$M_PFNMAP` indicates that the mapping should be done via *page frame number*[6] (see the description of vbn below). This is to ensure that the variables declared in a FORTRAN program are mapped correctly to the registers. The `SEC$M_WRT` mask bit is to allow the mapped section to have read and write access.

`gsdnam` — This is the global section name and is required only if the section is to be accessed globally. It is not used in the present case and is left blank.

`ident` — An argument containing match information for global sections. It only applies to global sections.

`pagecnt` — A longword containing the number of pages to be mapped. It is equal to 1 in the present case.

`vbn` — A long word containing the *page frame number* (`PFN`) of the memory where the mapped section begins. The `PFN` of any physical page in memory is contained in bits 9 through 29 of its physical address. The following paragraph shows how the `PFN` of the I/O page containing the addresses of the DRV11 registers may be obtained.

The address of the control and status register (CSR) of the DRV11 interface on the Q22-Bus I/O space is given as 767600_8 ($3EF80_{16}$)[4]. The first (least significant) 12 bits of this address, 7600_8 ($F80_{16}$), give the offset address in the Q22-Bus I/O space. This address must be added to the physical address at the beginning of the Q22-Bus I/O space, which is given on page H-3 of Reference [6] as 20000000_{16}, to obtain the physical address of the DRV11 CSR, i.e. $20000F80_{16}$. Of this physical address, bits 9 to 23 are the `PFN` and bits 0 to 8 are the byte offset within the page.

$$20000F80_{16} = \underbrace{10\ 0000\ 0000\ 0000\ 0000\ 111}_{\text{PFN}}\underbrace{1\ 1000\ 0000}_{\text{Byte offset}}{}_2$$

$$= 100007_{16} \qquad = 180_{16}$$

Therefore, the `PFN` of the mapped I/O page is 100007_{16}, and the byte offset of the DRV11 CSR is 180_{16}.

`prot` — A longword indicating the protection mask for the mapped section. This argument may be omitted and defaults to read and write access for all users.

Once the I/O page containing the CSR of the DRV11 interface has been mapped, access to its CSR and input and output buffers may be made via the addresses in the process's virtual memory corresponding to these register's offsets relative to the beginning of the mapped page. From the description of vbn previously, the byte offset of the DRV11 CSR is $180_{16} = 394_{10}$. The word (2 bytes) offset is then $394/2 = 192$.

[6] Page frame number is the address of the first byte of a page in physical memory.

Hence the CSR may be accessed via iopage(192), and the output and input buffers become iopage(193) and iopage(194) respectively.

Note that the input buffer and the high byte of the CSR of the DRV11 are read only. For some reason, the VMS system insists that the input buffer, iopage(194), be equated to a temporary variable before being output. Otherwise a hardware error (presumably attempting to write to a read-only address) occurs.

3.2 Connecting To Interrupts

Connecting a process to an interrupt vector allows the process to receive notification of interrupts from some hardware device. The notification may be via any combination of the following:—

- executing a user-supplied interrupt-service routine,
- setting an event flag in the calling process, or
- executing an Asynchronous System Trap (AST) service routine that gains control following the interrupt.

Before using the connect-to-interrupt system service, the hardware device must be configured using the SYSGEN CONNECT command of the VMS operating system. The following dialogue shows the procedure to configure the DRV11 as device OAA0 (OA is the standard VMS device name for the DR11C interface and VMS recognizes the present variant of DRV11 adapter only as a DR11C):—

```
$ SET DEFAULT SYS$SYSTEM
$ RUN SYSGEN
SYSGEN> CONNECT OAA0 /ADAPTER=0 /CSR=%0767600 /VECTOR=%0300 -
        /DRIVER=CONINTERR
SYSGEN> EXIT
$
```

The components of this command have the following meaning:—

- /ADAPTER=0 — has no significance on the MicroVAX, but must be included to conform to the syntax of the command.
- /CSR=%0767600 — specifies the Q22-Bus address of the DRV11 CSR.
- /VECTOR=%0300 — specifies the interrupt vector table address of interrupt 'A' on the DRV11 adapter.
- /DRIVER=CONINTERR — connects the DRV11 to a skeletal interrupt driver.

Note that all addresses (CSR and vector) are preceeded by %O to indicate that they are octal numbers. This SYSGEN CONNECT command must be executed every time the MicroVAX system is booted.

The previous command connects only one of the two interrupts of DRV11 to the CONINTERR driver. To connect the other one, the following command must also be executed in SYSGEN.

```
SYSGEN> CONNECT OAB0 /ADAPTER=0 /CSR=%0767600 /VECTOR=%0304 -
        /DRIVER=CONINTERR
```

where 304_8 is the interrupt vector table address associated with DRV11 interrupt 'B'.

Note that we have effectively defined two devices — OAA0: and OAB0: — both accessing the same physical device registers, but responding to different interrupts. Interrupt 'A' is used by the parallel data bus to notify the MicroVAX of completion of a data transfer. Interrupt 'B' is used to warn the MicroVAX of some error conditions.

At run time, a process wishing to be notified of interrupts generated by the DRV11 must first assign the DRV11 devices to associate a VMS channel with each device. This may be done via a system service call such as:—

```
status = SYS$ASSIGN ( '_OAA0:', chan, ,)
```

where chan (declared as an INTEGER*2 variable) will contain the value, returned by the function call, of the channel associated with the DRV11 device OAA0:. Both OAA0: and OAB0: should be assigned even if both interrupts are not to be used. Assigning a device prevents other processes from accessing it, and hence avoids problems with other processes altering the contents of the DRV11 registers.

The process can now be connected to the device interrupt via a call to the system service SYS$QIO in the following format:—

```
 status = SYS$QIO ( [efn], [chan], [func], [iosb], [astadr],
1         [astprm], [p1], [p2], [p3], [p4], [p5], [p6] )
```

Two calls to SYS$QIO, one for each interrupt (device) are required. All parameters except the p's have standard *SYS$QIO meanings* (see, for example, the chapter on *QIO System Service* in Reference [7]). The following are of specific interest:—

chan — The channel number obtained from a previous call to the SYS$ASSIGN routine.

func — A word containing the I/O function code — when connecting to an interrupt, this code may be either IO$_CONINTREAD or IO$_CONINTWRITE. The definitions of these function codes are contained in the system definition library module $IODEF which is included in the program source in the following manner:

```
INCLUDE '($IODEF)'
```

or the codes may be declared as EXTERNAL by:

```
EXTERNAL IO$_COINTWRITE, IO$_COINTREAD
```

p1 — A longword containing the address of a descriptor for a buffer containing code and/or data. It is not used in the present case.

p2 — A longword containing the address of an entry point list which is not used in the present case.

p3 — A longword containing flags, and event flag number specification. The flags are used to describe options for connect-to-interrupt facility, and are contained in the low-order word. The high-order word contains an event flag number to be set when an interrupt occurs. The relevant flags required in the present case are:--

`CIN$M_EFN` (value 1) indicates that an event flag is to be set on interrupt.

`CIN$M_REPEAT` (value 4) indicates that the process should be left connected to the interrupt vector until the connect is cancelled – usually by the process exiting.

The values of `CIN$M_...` are obtained from the macro expansion of the module `$CINDEF` which is contained in the VMS system macro library `LIB.MLB` in the `SYS$LIBRARY:` directory.

The only reliable way to set up this long word is to set up a FORTRAN record which maps the union of two words (`INTEGER*2`) with a single longword (`INTEGER*4`) as follows

```
STRUCTURE
    UNION
      MAP
        INTEGER * 2 switches
        INTEGER * 2 efnum
      END MAP
      MAP
        INTEGER * 4 flagword
      END MAP
    END UNION
END STRUCTURE
```

A variable of the above record type may then be created for each interrupt, for example, as:

```
RECORD  /flag/ dra_flags, drb_flags
```

The variable '`switches`' is initialized with the statements

```
dra_flags.switches = CIN$M_EFN .OR. CIN$M_REPEAT
drb_flags.switches = CIN$M_EFN .OR. CIN$M_REPEAT
```

and the event flag numbers, say 1 and 2, are specified as

```
dra_flags.efnum = 1          ! for event flag #1
drb_flags.efnum = 2          ! for event flag #2
```

p4 — The name of an AST routine (which must be declared as `EXTERNAL`) to be called as the result of an interrupt.

p5 — An AST parameter to be passed to the AST routine when an interrupt occurs.

p6 — The number of AST control blocks to be pre-allocated. This is not required in the present case.

Finally, to allow the interrupt signal to be received by the user program, the *interrupt enable bits* (INT ENB A and INT ENB B, see Figure 2) in the CSR of the DRV11 interface adapter must be set to 1. This may be achieved with the statements

```
iopage(dr_csr) = IBSET ( iopage(dr_csr), v_intenba )
iopage(dr_csr) = IBSET ( iopage(dr_csr), v_intenbb )
```

The index `dr_csr` refers to the location of DRV11 CSR in the mapped variable array `iopage`. The bit positions of INT ENB A and INT ENB B in the CSR are `v_intenba` and `v_intenbb` respectively. They are pre-defined via the declarations

```
INTEGER*2 dr_csr
INTEGER*2 v_intenbb, v_intenba
PARAMETER ( dr_csr = 192, v_intenbb = 5, v_intenba = 6 )
```

Thus, if the REQ A bit is asserted while INT ENB A is set in the DRV11 CSR, event flag number 1 will be raised (set). If the REQ B bit is asserted while INT ENB B is set, event flag number 2 will be raised.

3.3 Subroutine DR_INIT

The above procedures for setting up the interface between a process and the DRV11 at run time have been combined into a single routine `DR_INIT`. This routine (see Appendix D) does the following:—

1. Maps the Q22-bus I/O page containing the DRV11 registers to an array `iopage` in the process's virtual address space.
2. Assigns both "devices" and obtains their channel numbers.
3. Connects to both DRV11 device interrupts. Event flag number 1 is associated with the DRV11 interrupt A, and event flag number 2 is associated with DRV11 interrupt B.

The I/O page is mapped to an array `iopage` defined as:—

```
INTEGER*2 iopage(0:255)
COMMON /dr_common/iopage
```

This array, and hence the common block in which it is contained, must be page aligned. This may be done by including an option file `dr_link.opt` with any LINK [8] command (qualified with the LINK switch `/OPT`) which incorporates `dr_init`. The option file contains the line:

```
PSECT_ATTR = dr_common, page
```

4 SOFTWARE I/O INTERFACE PROCESS — DIGIO

4.1 Requirements Of The I/O Interface Process

Requests for data transfers to or from the parallel bus may arise under many different circumstances, and may occur as the result of the operation of many different processes. As the data acquisition software is currently set up, any of the following processes could require such transfers:—

1. COM??? (the many variants of compute sub-processes) will need to transfer information to the bus during real-time operations (e.g. set point Mach number, Reynolds number length scales etc.) and will obtain raw data from devices connected to the data bus via transfers from the bus.

2. MON??? (the many variants of monitor sub-processes) will need to acquire data from whatever module is being monitored. The frequency with which MON??? processes request such data will depend on what type of operation is being monitored (e.g. when monitoring the current state of health of a strain gauge balance more frequent requests will be made as the balance and/or model maximum loads are approached).

3. TST??? (variants of a general process used to test individual modules) will need to obtain data from the particular module under test as well as to write data to that module.

To coordinate the use of the DRV11 interface between the parallel bus and the MicroVAX, all input and output of data to or from the parallel data bus is carried out by a single process — DIGIO. This process will be run (detached) from the system startup file as:—

```
$ RUN /PROCESS=DIGIO /ERROR=DIGIO.ERR /UIC=[200,0] /DETACH -
      $DISK1:[DATAIN.EXE]DIGIO.EXE
```

and hence will be up and running whenever the MicroVAX is operational. The qualifier /ERROR=DIGIO.ERR directs any run-time error messages resulting from the execution of DIGIO to the file DIGIO.ERR. The /UIC=[200,0] qualifier assigns the DIGIO process a user identification code having a group number of 200, which is common to all users on the MicroVAX system, except the system manager and a few special accounts. This is to ensure that DIGIO can communicate with other processes run by other users (see section 4.6). $DISK1:[DATAIN.EXE] is the directory where DIGIO.EXE is located.

Consideration of the many possible processes requiring service from DIGIO shows that requests for data transfers from some processes need to be satisfied more quickly than others. For example, requests for the real-time COM??? processes must be dealt with immediately, to guarantee that raw data are gathered at the desired moment in time, or while the tunnel conditions remain as required. On the other hand, it matters little if the data requested by MON???, for example, are not available immediately. In fact, it is desirable that requests already in progress from processes such as MON??? be aborted in favour of a request from COM???. This suggests a multi-level priority system in which a low priority request already in progress makes way for a higher

priority request, and that low priority requests queued for DIGIO be passed over in favour of requests of higher priority. Most, if not all the features of an ideal system can be realized with a two-level — high and low — priority system.

Implicit in the above discussion is the provision of some sort of queueing system for requests to DIGIO. Requests queued to DIGIO would need to include the following information:—

1. The priority of the request.
2. The number of data transfers required to complete this request.
3. A list of addresses on the digital data bus to, or from, which data are to be transferred.
4. The location where data to be transferred are to be stored or found.

Communication between DIGIO and processes requesting data transfers will also need to include some mechanism for the notification of errors. In general, errors will be of one of three general types:—

1. A serious error may be signalled by the parallel data bus, indicating some sort of fault condition, either on the bus or in the interface, from which recovery is not possible.
2. The parallel data bus may signal the occurrence of a recoverable (non-fatal) error, for example a strain-gauge amplifier overloading. Operations may continue following such errors but data from that particular module on the bus may be invalid.
3. A low-priority request may either be aborted or ignored in favour of a higher priority request. In either case, the data or action expected by the requesting process will be invalid, and the process must be notified so that it may take the appropriate action (in many cases this will simply be to requeue the request).

The operation of DIGIO thus requires effective communication and synchronization with other processes wanting to access the DRV11 interface, or for that matter, the data bus. This has been achieved by making use of the following programming techniques provided by VMS and VAX FORTRAN:—

1. **Mailboxes:—** These are channels provided by VMS for interprocess communications (see section 3.4.2 of Reference 5). Messages are passed into mailboxes in the form of character strings. They are queued within the mailbox and are read by the destination process sequentially.
2. **Installed Common Blocks:—** A common block defined in a FORTRAN program unit may be installed in the system memory as a shareable image. A program wanting access to this common block, as well as defining and declaring it within the program unit, must also be linked to this image (see section 4.4.4 or page 3-46 of Reference 5).
3. **Common Event Flags:—** Event flags are used to signal either the status (success or failure) of an operation, or the occurrence of a particular event.

They may be *waited for* when one operation must be completed before the next one can continue, or they may be *checked* to see if they have been set as an indication of a status or condition.

4.2 The DIGIO Mailbox

The primary means of communication between DIGIO and other processes is a queue containing messages requesting DIGIO to carry out data transfers. This queue is realized as a VMS mailbox called `digio_mbox`. DIGIO creates this mailbox during its initialization phase via the statement

```
 status = sys$crembx ( %VAL(1),
1                      %VAL(mbox_chan),
2                      %VAL(256),
3                      %VAL(1024), , ,
4                      'digio_mbox'     )
```

where

- `%VAL(1)` — indicates that the mailbox is permanent. This ensures that the mailbox logical name is entered into the system logical name table (rather than the process logical name table) making it available to all other processes.
- `mbox_chan` — is a word (`INTEGER*2`) which on return from `SYS$CREMBX` will contain the channel number assigned by VMS to the mailbox.
- `%VAL(256)` — indicates that the length of any message sent to this mailbox will not exceed 256 bytes (or 256 characters in VAX FORTRAN).
- `%VAL(1024)` — specifies the size (in bytes) of a buffer set aside by VMS for temporarily storing queued messages.
- `'digio_mbox'` — is the name given to the mailbox.

Any process wishing to queue a request to DIGIO should first assign a channel number to the mailbox via the statement

```
status = SYS$ASSIGN ( 'digio_mbox', digio_mbox_chan, , , )
```

where `digio_mbox_chan` (`INTEGER*2`) is the returned channel number and is used for all subsequent transfers of messages to the mailbox. The requesting process may then send a message to `digio_mbox` via the system service routine `SYS$QIOW` of the following form.

```
 status = SYS$QIOW ( , %VAL(digio_mbox_chan),
1                      %VAL(mbox_write_code), , , ,
2                      %REF(digio_message),
3                      %VAL(length) , , , ,       )
```

where

- `digio_mbox_chan` — is the channel number returned by `SYS$ASSIGN`.

- mbox_write_code — is a longword (INTEGER*4) containing the SYS$QIOW function code "write to a mailbox". This is set via

```
mbox_write_code = IO$WRITEVBLK .OR. IO$M_NOW
```

  where the modifier IO$M_NOW specifies that the requesting process should complete the function call *now* rather than waiting for DIGIO to read the message, and IO$WRITEVBLK is the mailbox write function. All the IO$... function codes are defined in the system definition library module $IODEF of the system library file SYS$LIBRARY:FORSYSDEF.TLB. The $IODEF module may be included in the source code of the user written program with the statement:—

```
INCLUDE '($IODEF)'
```

- digio_message — is the message to be sent to DIGIO (see description below).
- %VAL(length) — specifies the length of the message.

DIGIO reads messages queued in digio_mbox via the following SYS$QIOW call

```
 status = SYS$QIOW ( , %VAL(mbox_chan),
1                    %VAL(mbox_read_code),
2                    mbox_iosb, , ,
3                    %REF(mbox_message),
4                    %VAL(256)            )
```

where all variables are similar to those in the previous SYS$QIOW call except that the function code is given by

```
mbox_read_code = IO$READRBLK .OR. IO$M_NOW
```

The modifier IO$M_NOW specifies that the SYS$QIOW function is to be completed *now* rather than waiting for the mailbox to contain a message. The input/output status block mbox_iosb is specified by the structure:—

```
STRUCTURE /iostat_block/
  INTEGER*2 iostat, msg_length
  INTEGER*4 sender_pid
END STRUCTURE

RECORD /iostat_block/ mbox_iosb
```

In addition to the status of the data transfer, DIGIO can also obtain the length of the message read, and the identity of the process which sent the message from this status block.

4.3 The DIGIO Mailbox Message

Experiments with the MicroVMS system have indicated that when writing to or reading from a mailbox a few long messages incur significantly less overhead than many short messages. Hence, if a process requires several data transfers, it is better

to combine them into one request, rather than to have a series of requests each consisting of only one transfer. This arrangement has some disadvantages, the most important being the loss of a one-to-one relationship between bus errors and data transfer requests, but in normal circumstances where the error rate is low, the faster overall rate of data transfer will outweigh this disadvantage.

Messages sent to `digio_mbox` are character variables with the following contents:—

- `priority` — indicates the priority of the request — 'H' for high, 'L' for low.
- `requestor_index` — identifies the requesting process.
- `num_addresses` — the number of addresses to or from which data is to be read or written. A maximum of 50 addresses (and hence data transfers) may be included in each request.
- `addresses(i), i = 1, num_addresses` — indicates the actual bus addresses to or from which data is to be read or written.

These variables are encoded into the character array variable `mbox_message` via an internal write statement with a format of:—

```
FORMAT ( A1, 1X, I1, 1X, I2, <num_addresses>( 1X, I4 ) )
```

All information needed by DIGIO to complete the transfers is contained in this message, except that there is no indication of the direction of each data transfer, i.e. whether it is a read or write, as this is encoded directly in the bus address for each data transfer. The parallel data bus follows the convention that all address to which data may be written are *odd* (least significant bit set) while those which may be read are *even* (least significant bit not set). Hence, it is possible to include a mixture of reads and writes in a single request to DIGIO.

In addition to detecting whether an address is odd or even, DIGIO detects three *special* addresses indicating a request for a non-standard operation on the bus. These are:—

- 0000 — DIGIO treats this address as a NOP (no operation) and does not process the address further. The need for such an address arises when a programmer wishes to reserve a "space" in `digio_message` for a future bus operation that is yet to be developed.
- 0001 — DIGIO sends a "trigger all analogue to digital converters (ADCs)" to the bus to begin a sample of the current data values.
- FFFF — DIGIO sends a "master reset" to the bus. This code must be used with *extreme* care (see further comments in section 4.7).

In addition to sending the above message to the `digio_mbox` mailbox, a requesting process must also notify DIGIO that a message has been sent. This is achieved by setting an event flag (see section 4.6) — number 65 for a low priority request, or number 64 for a high priority one.

4.4 The DIGIO Global Common Block

To coordinate the concurrent use of DIGIO by several processes, and to provide an efficient means of communicating data between requesting processes and the parallel bus, a global common area called `digio_common` is created. The data structures contained in this common area are maintained either by DIGIO itself or by processes using DIGIO. The common block and the variables which are contained in it are defined in an include file `DIGIO_INCLUDE.FOR` (Appendix B). The data structures, their purpose, and their usage are described in the following sub-sections.

4.4.1 Bus Module Usage Tables

Wind tunnel data gathering devices connected to the parallel bus are known as "instrumentation modules" or just "modules". Each module is electrically distinct and has one and only one physical connection to the bus. Modules usually coordinate all data transfers associated with either a single device or a logical grouping of tunnel instrumentation. For example, there is a strain gauge balance module incorporating six strain-gauge amplifiers, their analogue to digital converters and their associated controls, which coordinates all data transfers to or from a six-component strain-gauge balance. Other modules are associated with Scanivalves, tunnel parameters, etc. In its present form, the parallel data bus can handle up to 16 modules (numbered in hexadecimal from 80 to 8F), but this could be extended if the need should arise. The current allocation of modules to module numbers is included in Table 1.

Table 1: Allocation of module number to data acquisition modules.

Module Number	Module Name
80	Auxiliary 6-Channel AC Amplifiers Module
81	Strain Gauge Module with 6-Channel AC Amplifiers
83	Scanivalve Module with 6-Channel DC Amplifiers
85	VPT Tunnel Parameter Module
87	LSWT Tunnel Parameter Module
8A	Inclinometer Module
8B	Actuator Module

Two variables in `digio_common` keep track of those modules which are electrically connected to the bus. The first variable, `module_table`, is a logical array defined as

```
LOGICAL       module_table(0:15)
```

in which each element is set to `.TRUE.` if the module with the corresponding number is electrically connected to the bus. The second variable, `module_names`, defined as

```
CHARACTER*80 module_names(0:15)
```

is a similarly dimensioned array of character variables each with a length of 80 characters. If an element of `module_table` is `.TRUE.`, the corresponding element of `module_names` will contain a character string which identifies that module.

These two variables are initialized by DIGIO when it is started, and are updated whenever DIGIO receives a master reset command. In either case, DIGIO attempts to read a special address (8x60, where x is the module number in hexadecimal, see section 4.5) contained in each module which returns that module's identification string. If the string is not returned after 10 milliseconds, DIGIO times out and sets the corresponding entry in `module_table` to `.FALSE.`.

Processes requesting data transfers via DIGIO may use the information in this data structure, typically to ensure that all modules required for their particular data transfers are present, but they *must not* change any of the information contained in these variables.

4.4.2 Data Arrays And Usage Indicators

When writing or reading data to or from the parallel bus, DIGIO obtains data to be written and stores data which has been read in an array in the global common area defined as

```
INTEGER*2  data_list(1000)
```

Each process transferring data via DIGIO is allocated one or more blocks of 50 elements within this array. In this way, the data belonging to one process using DIGIO remains independent of all others. To keep track of which process owns which block of elements within `data_list` and to determine where each process's data are stored, DIGIO maintains the following data structure

```
INTEGER*2  num_requestors
LOGICAL    requestor_index_table(5)
LOGICAL    data_usage_table(20)
```

The variable `num_requestors` contains the number of processes currently using DIGIO for transfers to, or from, the parallel bus — the number of processes "connected" to DIGIO. Currently, the maximum number of processes which may be connected to DIGIO at any one time is set (somewhat arbitrarily) to 5. Whenever a process wishes to connect to DIGIO it must first check that `num_requestors` is less than 5, and if so, increment `num_requestors`. When a process disconnects from DIGIO, `num_requestors` must be decremented.

Having incremented `num_requestors`, a process connecting to DIGIO must then find the first available empty (i.e. `.FALSE.`) element in `requestor_index_table` and set it to `.TRUE.`. The index of this element then becomes that process's unique identifier (requestor index) in DIGIO and is used by DIGIO to derive several quantities associated with that process. The connecting process will also require the value of the requestor index and it should therefore also store it locally. When disconnecting from DIGIO a process should return that element of `requestor_index_table` to `.FALSE.`.

A connecting process must then allocate one or more blocks of 50 elements in the array `data_list` for its own use. This is done by searching the `data_usage_table` for an empty (i.e. `.FALSE.`) element and setting that element to `.TRUE.`. If the process requires more than 50 elements of `data_list`, the process must ensure that as many subsequent elements of `data_usage_table` as the number of blocks (of 50 elements

in data_list) required are also empty, and set each one to .TRUE.. In other words, the elements of data_list allocated to each *must* be contiguous.

In addition to the above variables defined in the DIGIO global common block, an integer variable data_usage_index, defined locally within the connecting process, is used to store the first entry of data_usage_table that is allocated to the process so that when it disconnects from DIGIO, each element of data_usage_table set by that process may be returned to .FALSE..

Another requirement of a process connecting to DIGIO is to determine the index of the address in data_list where data for this process begins. This index is stored in an integer variable data_start_addr, also defined locally within the connecting process, and may be calculated via a statement of the form:—

```
data_start_addr = (data_usage_index - 1) * 50 + 1
```

4.4.3 Display Process Activity Indicator

The display process activity indicator variable, defined by

```
LOGICAL      dspon
```

is used to indicate that a display sub-process (DSP???) is currently active. It is not part of the DIGIO data structure, and is only included in the global common area for convenience (this common is available to *all* processes connected to DIGIO). It is used by various processes to determine whether or not certain computations which are only required when a display process is active should be done. (This is necessary because display processes may be started and stopped asynchronously without reference to other processes.)

All display sub-processes must ensure that dspon is set to .TRUE. when they are started, and returned to .FALSE. when they are stopped (so long as no other display sub-process remains active).

4.4.4 Linking To The Global Common Block

For the global common block to be accessible by other processes, it must be installed in system memory as a shareable image. The program, DIGIO_INSTALLED.FOR, was created to install digio_common as a global common block. It contains the following three statements:—

```
PROGRAM DIGIO_INSTALLED
INCLUDE '[DATAIN.DIGIO]DIGIO_INCLUDE.FOR/LIST'
END
```

The program is compiled and linked as follow:—

```
$ FORTRAN DIGIO_INSTALLED
$ LINK/SHAREABLE DIGIO_INSTALLED
```

The file protection on the executable code, DIGIO_INSTALLED.EXE needs to be modified so that the "world" has *read* and *write* access to it. This is achieved by the command:—

```
$ SET PROTECTION:W=RW DIGIO_INSTALLED.EXE
```

To install the image (the executable code of a program), CMKRNL privilege is required. The following steps show how this may be done:—

```
$ SET PROCESS/PRIVILEGE=CMKRNL
$ INSTALL CREATE $DISK1:[DATAIN.EXE]DIGIO_INSTALLED.EXE -
  /SHARED/WRITEABLE
$ SET PROCESS/PRIVILEGE=NOCMKRNL
```

$DISK1:[DATAIN.EXE] is the directory where DIGIO_INSTALLED.EXE is located.

Any process that wishes to transfer data to, or from, the parallel bus via DIGIO should include the global common block in its source code via the statement

```
INCLUDE '[DATAIN.DIGIO]DIGIO_INCLUDE.FOR'
```

When linking the object code, an option file containing the line

```
$DISK1:[DATAIN.EXE]DIGIO_INSTALLED/SHAREABLE
```

must be included in the LINK command with the /OPT qualifier. The following dialogue shows how DIGIO.EXE is built (DIGIO.OPT being the option file):—

```
$ FORTRAN DIGIO.FOR
$ LINK/EXE=DIGIO.EXE DIGIO.OBJ,DIGIOLIB.OLB/LIB,DIGIO.OPT/OPT
```

DIGIOLIB.OLB is the object library containing the subroutines used by DIGIO.

4.5 Module Addresses

Each module number on the parallel bus is assigned 256 (100_{16}) addresses of the form $8x??_{16}$, where x is the module number in hexadecimal. Thus, module number 5 is assigned addresses from 8500_{16} to $85FF_{16}$. However only those addresses above $8x60_{16}$ (i.e. a total of 160 ($A0_{16}$)) are valid external read/write addresses.

To avoid the need for system software developers to be aware of the hexadecimal addresses of each function on each module, all valid parallel bus addresses have been mapped to an array defined by

```
INTEGER*2     address_list(0:1599)
```

Elements of address_list are defined in the file BUS_ADDRESSES.FOR and are included in the main program unit of DIGIO at compilation time. Referencing this array, rather than the actual bus addresses, allows system programs to be written so that they are independent of changes in the allocation of bus addresses within modules.

The three special bus addresses — *No operation* (NOP), *master reset*, and *trigger* — are contained in address_list(0), address_list(1) and address_list(2) respectively.

Appendices E and F give full listings of the allocation of bus addresses to the array address_list. Because of differences in module construction and usage in the two wind tunnels, bus address allocations in the two tunnels are different.

4.6 Data Transfer Complete/Error Notification

When DIGIO has completed processing a data transfer request, it will either:

1. set an event flag indicating that the transfer was completed successfully (the success event flag) or,
2. set an event flag indicating that the transfer failed or was incomplete (the failure event flag). There are two reasons for the failure event flag to be set:—
   - the data bus failed to respond within a timed-out period (50 millisecond), or
   - at the completion of a low-priority request which has been either aborted, or passed over, in favour of a high-priority one.

A group of event flags, or a common event flag cluster, has been specifically allocated to be shared and used among processes[7] communicating with DIGIO. This event flag cluster is created with the SYS$ASCEFC system service. A process can reference this cluster by invoking the same SYS$ASCEFC system service and specifying the same cluster name. The DIGIO event flag cluster is given the name of `dicluster` and is assigned to a character array variable, `digio_efn_cluster`. Note that although VAX FORTRAN is a case insensitive language, the event flag cluster name is case sensitive. Thus to create or associate a process with the DIGIO event flag cluster the following code fragment is used.

```
CHARACTER*9  digio_efn_cluster
PARAMETER    ( digio_efn_cluster = 'dicluster' )

status = SYS$ASCEFC ( %VAL(64), digio_efn_cluster , , )
```

The common event flag cluster thus created consists of the 32 event flags from 64 to 95 inclusive.

As discussed above, most errors which occur on the parallel bus will be informative rather than fatal. Typically, these errors will indicate that an input transducer (such as a strain gauge amplifier) has been, for example, over-ranged. Generally such events will not occur synchronously with requests for data transfers. Therefore, the state of the error indicating interrupt (REQ A bit) on the DRV11 is continuously monitored via an Asynchronous System Trap (AST) routine. Whenever an error is detected, control is transferred to the AST routine (called `DIGIO_ERROR_AST`), which determines the type and source of the error. It interrogates the parallel bus and creates an error message containing the module number in which the error occurred, an indication of whether or not the error is fatal (in which case the parallel bus would have to be sent a *master-reset* before further data transfers are attempted), and a description of the error. The AST routine sends this message to a mailbox — the error mailbox — associated with each process currently connected to DIGIO. The error message is packed into a single character variable 40 characters long, the individual components being available via an internal read statement, as shown in the following code fragment:—

[7] Processes can share a common event flag cluster only if they have the same *group number* in their user identification code (UIC), i.e. they are executed by users whose UIC has the same group number.

```
      CHARACTER error_message*40, message_text*37, severity*1
      INTEGER*2 module_number

      READ (error_message,10) severity, module_number, message_text
10    FORMAT ( A1, I2, A37 )
```

The AST routine will send a copy of the error message to mailboxes with names created by the concatenation of `'error_mbox_'` with the requestor index of each process currently connected to DIGIO. Hence, to receive an error message, each process must create a mailbox with a name of the form `error_mbox_?`, where ? is the requestor index of the process. The AST routine `DIGIO_ERROR_AST` then sets an *error event flag* associated with each connected process to indicate that an error message has been sent.

To allow all processes connected to DIGIO to proceed independently, DIGIO maintains a separate set of event flags — `efn_success` (*success event flag*) ,`efn_failure` (*failure event flag*) and `efn_error` (*error event flag*) — for each connected process. DIGIO derives the event flag numbers allocated to each of these functions from each connected process's requestor index, `requestor_index`, via code of the form

```
efn_success = 63 + 3 * requestor_index
efn_failure = 64 + 3 * requestor_index
efn_error   = 65 + 3 * requestor_index
```

which allocates 3 consecutive event flag numbers for each connected process in the range of 66 to 80. Each process connecting to DIGIO should therefore compute the event flag numbers allocated for its use via code similar to that above.

It should be noted that the above event flags numbers (66 to 80), together with numbers 64 and 65 used by connected processes when sending request messages, should not be used for other purposes.

4.7 Master Reset

As mentioned above, a reference to `address_list(0)` (which translates to a bus address of $FFFF_{16}$) in a data transfer request causes DIGIO to send a master reset to the parallel bus. This should be used with extreme care since it destroys (initializes) all data stored in all modules connected to the bus. However, a master reset is the only way to initialize the parallel bus properly at power up, to reset the bus to a known state whenever a module is electrically connected to or disconnected from the bus, or to reinitialize the bus following a fatal (non-recoverable) error.

The first situation is taken care of by DIGIO. The other two are the responsibility of system processes. However, before sending a master reset request to DIGIO and hence to the bus, a process *must* be the *only* process connected to DIGIO — any other process would have no knowledge of the change to the data on the bus. A process which wishes to initiate a master reset must therefore ensure that `num_requestors` is equal to one, indicating that only one process (itself) is connected to DIGIO.

5 SUMMARY OF REQUIREMENTS FOR A PROCESS COMMUNICATING WITH THE BUS VIA DIGIO

The following sections summarize the actions which must be taken by a process that needs to transfer data to or from the parallel bus via DIGIO. Many of these have been at least implied in the descriptions of how the various parts of the system operate in previous sections. However, in the following sections, the actions are grouped in a logical order, and code fragments to achieve each action are provided.

5.1 Definition Of Variables

Before communicating with DIGIO, the data structure must be set up correctly for the process. The following are definitions of some of the more important variables.

1. All system routines *must* be defined as INTEGER*4 variables before being used. Some of these routines are

```
 INTEGER*4    status,
1             SYS$CREMBX,
2             SYS$ASSIGN,
3             SYS$ASCEFC,
4             SYS$DACEFC,
5             SYS$SETEF,
6             SYS$CLREF,
7             SYS$WFLOR,
8             SYS$QIOW
```

   The variable **status** is widely used to hold the return status code of the system routines, and must be defined as an INTEGER*4 also.

2. Two condition codes defined in the system object and shareable image libraries used in the process's code must be made known to the process. The easiest way to do this is to define the condition codes as external symbols, thus

```
 EXTERNAL     ss$_wasset
 EXTERNAL     ss$_endoffile
```

   The codes may then be referred to by using the built-in function %LOC which returns the address of its argument.

3. Several variables must be available to all program units (routines) in the process. These would normally be included in a common block shared by all program units. These variables are:—

```
 CHARACTER    digio_message*256
 INTEGER*2    digio_mbox_chan,
1             error_mbox_chan

 INTEGER*4    requestor_index,
1             data_start_addr,
```

```
2               efn_success,
3               efn_failure,
4               efn_error,
5               efn_mask
```

4. The remaining group of variables need only be defined in the program unit in which they are used. These should be defined as follows:—

```
 CHARACTER      error_mbox_name*12,
1               error_message*40

 CHARACTER      digio_mbox_name*(*)
 PARAMETER      ( digio_mbox_name = 'digio_mbox' )

 CHARACTER      digio_efn_cluster*(*)
 PARAMETER      ( digio_efn_cluster = 'dicluster' )

 INTEGER*4      mbox_write_code
 PARAMETER      ( mbox_write_code = '70'X )

 INTEGER*4      mbox_read_code
 PARAMETER      ( mbox_read_code = '71'X )
```

   Note the use of **PARAMETER** statements to define several constants. The disadvantage of not being able to include such variables in **COMMON** blocks is outweighed by the protection given to their values — any attempt to change the value of a constant defined via a **PARAMETER** statement will produce an error at compilation time.

There are two *include* files designed to facilitate the creation of the above data structure. They are **DIGIO_INCLUDE.FOR** and **DIGIO_DEFS.FOR**, both of which reside in the directory **$DISK1:[DATAIN.DIGIO]**.

DIGIO_INCLUDE.FOR defines the global common block, **DIGIO_COMMON**, and its associated variables (section 4.4). It must be included in each program unit (subroutines and functions) which references any one of these variables. A listing of this *include* file is given in Appendix B. Note that when linking, an option file must be included to link to this shared, shareable common (see section 4.4.4).

DIGIO_DEFS.FOR defines the standard variables (groups of variables referred to in items 3 and 4 above) required by a process when communicating with DIGIO. It defines the variables common (but defined locally within the process) to all requesting processes such as **digio_mbox_name**, **low_priority_efn** and **high_priority_efn**. It also defines process-specific variables such as **digio_mbox_chan**, **error_mbox_name**, **requestor_index** and so on. These process-specific variables are grouped into a common block named **process_common** so that they may be used by all the routines within the process. Appendix C contains a listing of this file.

These files are included into the appropriate program units with the statements

```
INCLUDE '$DISK1:[DATAIN.DIGIO]DIGIO_INCLUDE.FOR'
INCLUDE '$DISK1:[DATAIN.DIGIO]DIGIO_DEFS.FOR'
```

and the appropriate variables will be defined and set up accordingly.

5.2 Connecting To DIGIO

Before a process can transfer data to or from the parallel bus, it must "connect" to DIGIO via the following steps:—

1. Associate the DIGIO common event flag cluster with

```
status = SYS$ASCEFC ( %VAL(64), 'dicluster' )
```

2. Connect to the DIGIO mailbox by

```
status = SYS$ASSIGN ( digio_mbox_name, digio_mbox_chan, , , )
```

3. Increment the number of processes connected to DIGIO via

```
num_requestors = num_requestors + 1
```

   Note that the value of **num_requestors** should be checked before incrementing and if it is greater than, or equal to 5, the connection cannot be made until one or more processes disconnects from DIGIO.

4. Determine the process's requestor index via

```
i = 1
DO WHILE ( requestor_index_table ( i ) )
    i = i + 1
END DO
requestor_index_table ( i ) = .TRUE.
requestor_index = i
```

   The above code locates the first empty (**.FALSE.**) element in the array **requestor_index_table**, sets it to **.TRUE.** and remembers the value as **requestor_index**.

5. Create the process-specific error mailbox via

```
      error_mbox_name(1:11) = 'error_mbox_'
      WRITE (error_mbox_name(12:12),10) requestor_index
10    FORMAT ( I1 )
      status = SYS$CREMBX ( %VAL(1),
     1                      error_mbox_chan,
     2                      %VAL(40), , , ,
     3                      error_mbox_name )
```

6. Determine a starting address for the process's data storage area in **data_list**. First locate the first empty (FALSE.) element in the array **data_usage_table** via

```
i = 1
DO WHILE ( data_usage_table(i) .AND. i .LE. 20 )
    i = i + 1
END DO
```

If this process requires 50 or less locations in data_list (which will be the case for most of the processes connecting to DIGIO), the area in data_list pointed to by this element of data_usage_table will be sufficient. The connecting process must now mark this element in data_usage_table as being used and add the computed starting address to data_start_addr. The index of the first entry in data_usage_table is stored in data_usage_index via

```
data_usage_table(i) = .TRUE.
data_usage_index = i
data_start_addr = ( i - 1 ) * 50 + 1
```

If this process requires more than 50 locations in data_list, then it must ensure that as many subsequent elements of data_usage_table as the number of blocks (of 50 elements in data_list) required are also free (.FALSE.). Otherwise, the procedure is as set out above.

In both cases, if less than the required number of elements in data_usage_table is available, the connection must be aborted.

7. The event flag numbers allocated to this process must be computed from the value of requestor_index via

```
efn_success = 63 + requestor_index * 3
efn_failure = 64 + requestor_index * 3
efn_error   = 65 + requestor_index * 3
```

To allow the process to detect the setting of either the success or failure event flag via the SYS$WFLOR system service routine, an event flag mask can be created which is the logical OR of the success and failure event flag numbers. This may be achieved as follows

```
efn_mask = 0
efn_mask = IBSET ( efn_mask, MOD ( efn_success, 32 ) )
efn_mask = IBSET ( efn_mask, MOD ( efn_failure, 32 ) )
```

The above procedures have been included in the subroutine CONNECT_DIGIO with the standard variables defined as in DIGIO_DEFS.FOR. The calling syntax of this routine is

```
CALL CONNECT_DIGIO ( digio_status )
```

where digio_status is an INTEGER*4 variable (pre-defined in DIGIO_DEFS.FOR) returned from CONNECT_DIGIO indicating if the connection is successful (1) or not (0).

This subroutine is contained in the object library DIGIOLIB.OLB located in the directory $DISK1:[DATAIN.DIGIO], and may be linked to the executable code of a user program with the LINK qualifier $DISK1:[DATAIN.DIGIO]DIGIOLIB.OLB/LIB.

5.3 Transferring Data Via DIGIO

Once a process has successfully connected to DIGIO it may read or write data from or to the parallel bus by sending request messages to the digio mailbox.

Whether the request is for a read or a write, the first part of the DIGIO message remains the same, varying only with the priority of the request. It is therefore useful to generate this "header" part of the message once, and then to use it as part of all subsequent messages. This may be done via

```
      digio_message(1:1) = 'H'
      WRITE (digio_message(2:3),10) requestor_index
10    FORMAT ( 1X, I1 )
```

which generates a high priority request message. Substitution of an 'L' for the 'H' achieves the same for a low priority request.

The remainder of the message is dependent on the particular data to be transferred. If, for example, six variables (e.g. the outputs of six analogue to digital converters) are to be transferred from the sting balance strain gauge module, the remainder of the message could be generated via

```
      digio_message(4:36) = '  6  102  103  104  105  106  107'
```

which indicate that six addresses are to be transferred and includes the indices of the required elements of address_list. These values are defined in the file BUS_ADDRESSES.FOR which is used by DIGIO as a look up table for the appropriate addresses to be sent to the data bus.

The request message is sent to the DIGIO mailbox via the system service routine SYS$QIOW:—

```
      status = SYS$QIOW ( , %VAL(digio_mbox_chan),
     1                    %VAL(mbox_write_code), , , ,
     2                    %REF(digio_message),
     3                    %VAL(256), , , ,              )
```

The requesting process should then wait for DIGIO to set either the success or failure event flag to indicate that the request has been completed, determine which event flag was set, and process the data as required.

The subroutine DIGIO_SEND is designed to simplify the process of constructing the digio_message and the SYS$QIOW calling sequence. To request a data transfer, it is only necessary to put the required address indices into the array addr_index and invoke the routine with the following syntax:

```
      CALL DIGIO_SEND ( num_addresses, addr_index, digio_status )
```

where

- num_addresses — is an INTEGER*4 variable (pre-defined in DIGIO_DEFS.FOR) indicating the number of addresses to be sent,
- addr_index — is an INTEGER*2 array (pre-defined in DIGIO_DEFS.FOR) containing the indices of the addresses to be sent, and
- digio_status — is an INTEGER*4 variable indicating the return status of the routine. If digio_status=1 the operation is successful. If digio_status=0 the operation fails, and if digio_status=-1 an error has occurred.

The object code of DIGIO_SEND is found in $DISK1:[DATAIN.DIGIO]DIGIOLIB.OLB and may be linked to a user's program in the usual manner.

Note that the values read from the parallel bus are stored by DIGIO in data_list, beginning at the start of the requesting process's area, i.e. at data_start_addr, and in the same order as they appear in the request message or in addr_index. The values are returned in data_list as INTEGER*2 variables, and are able to represent the full 16 bit precision generated by the analogue to digital converters used on the parallel bus. Outputs from the analogue to digital converters are returned by DIGIO as offset binary (0000_{16}=$+32768_{10}$, $7FFF_{16}$=0_{10}, and $FFFF_{16}$=-32767_{10}). To convert the values in data_list from this form to the two's complement binary used by the MicroVAX (8000_{16}=-32768_{10}, 0000_{16}=0_{10} and $7FFF_{16}$=$+32767_{10}$), the following code fragment may be used.

```
INTEGER*2    mask
PARAMETER  ( mask = '7FFF'X )

adc_value(i) = IIEOR ( data_list(j), mask )
```

Note that there are two "reserved" words in the range returned by DIGIO; 0000_{16} or null, and $FFFF_{16}$ which if returned indicates that the data are not valid.

To write data to the bus, the procedure is similar, with the obvious exception that the data to be written must reside in data_list before the request message is sent. For example, to send a value to the bus to be used as a reference length for the Reynolds number calculations (for which the bus address resides in address_list(522)), the following code could be used.

```
data_list(data_start_addr) = renlen / 1000.
digio_message(8:15) = '  1  522'

status = SYS$QIOW ( , %VAL(digio_mbox_chan),
1                    %VAL(mbox_write_code), , , ,
2                    %REF(digio_message),
3                    %VAL(256), , , ,              )

status = SYS$SETEF ( %VAL(64) )
status = SYS$WFLOR ( %VAL(64), %VAL(efn_mask) )
status = SYS$CLREF ( %VAL(efn_failure) )

IF ( status .EQ. %LOC(ss$_wasset) ) THEN

C       Failure flag was set - do whatever is necessary

END IF
```

If the subroutine DIGIO_SEND is employed, the code fragment would then read:—

```
data_list(data_start_addr) = renlen / 1000.
num_addresses = 1
addr_index(1) = 522
```

```
      CALL DIGIO_SEND( num_addresses, addr_index, digio_status )
      IF ( digio_status )
C         Operation successful!
      ELSE
C         Operation failed!
      END IF
```

5.4 Treating Errors

Whenever an error occurs on the parallel data bus, DIGIO sends an error message to the specific mailbox associated with each process connected to DIGIO (this mailbox was created in step 6 of the connection process, described in section 5.2, or in the DIGIO_CONNECT routine). Whenever DIGIO sends such a message it notifies each connected process by setting its error event flag. Bus errors may occur at any time, and if not read by a connected task, will simply accumulate in each process's error mailbox. Thus whenever it is important for the process to be aware of possible errors, the error event flag should be checked and, if set, control transferred to a routine which reads and decodes the error message. This may be achieved via code such as

```
      status = SYS$CLREF ( %VAL(efn_error) )
      IF ( status .EQ. %LOC(ss$_wasset) ) THEN
          CALL TREAT_ERROR
      END IF
```

It is important that connected processes check the error event flag after each data transfer (to ensure the validity of data sent or received), and also before sending a request message. If a fatal error has occurred since the error event flag was last checked, any further messages sent to DIGIO will not be treated correctly, and at worst, DIGIO may *never* reply to the message, thus hanging the requesting process.

The routine used to read and decode error messages must allow for the possibility that more than one error message has accumulated in the error mailbox. One approach to this would be as follows:—

```
      STRUCTURE /iostat_block/
         INTEGER*2 status, msg_length
         INTEGER*4 sender_pid
      END STRUCTURE

      RECORD /iostat_block/ mbox_iosb
      CHARACTER error_message*40
      EXTERNAL  ss$_endoffile

      mbox_iosb.status = 1
      DO WHILE ( mbox_iosb.status .NE. %LOC(ss$_endoffile) )
C ***** Read the Mailbox until no more messages
          status = SYS$QIOW ( , %VAL(error_mbox_chan),
     1                          %VAL(mbox_read_code),
     2                          mbox_iosb, , ,
     3                          %REF(error_message),
```

```
     4                          %VAL(40), , , ,        )
          IF ( mbox_iosb.status .EQ. %LOC(ss$_endoffile) ) RETURN
C ***** Decode the error message and treat accordingly
      .
      .
      .
      END DO
```

The external variable **ss$_endoffile** is the system service status code indicating that there are no more messages in the mailbox.

5.5 Disconnecting From DIGIO

Before a process which has connected to DIGIO exits, it must return the DIGIO data structures to their original state. This may be done using the following steps.

1. Return this process's entry in **requestor_index_table** to **.FALSE.** i.e.

   ```
   requestor_index_table (requestor_index) = .FALSE.
   ```

2. Return this process's entry in **data_usage_table** to **.FALSE.** i.e.

   ```
   data_usage_table (data_usage_index) = .FALSE.
   ```

3. Decrement **num_requestors**.

4. Disassociate the DIGIO common event flag cluster via

   ```
   status = SYS$DACEFC ( %VAL( 64 ) )
   ```

5. Delete the error mailbox associated with this process:

   ```
   status = SYS$DELMBX ( %VAL( error_mbox_chan ) )
   ```

The process is then free to exit. The subroutine **EXIT_DIGIO** is provided to perform the above tasks and is called by

```
CALL EXIT_DIGIO
```

EXIT_DIGIO is contained in the object library **$DISK1:[DATAIN.DIGIO]DIGIOLIB.OLB**.

6 CONCLUSIONS

A software interface, DIGIO, has been developed for the new data acquisition system in the two main wind tunnels at ARL. It has been developed on a DEC MicroVAX II computer equipped with a DRV11 parallel I/O interface adapter. Access to the three registers of the DRV11 adapter is provided from the software by direct mapping of the Q22-Bus I/O page to program variables. This method produces a fast and efficient means of communicating with the parallel data bus via the DRV11 interface.

DIGIO handles all the instrumentation control and data transfer requests from various data acquisition processes. Up to five processes may access the parallel data bus at one time, which provides great flexibility for developing data acquisition software. Details have been provided of the steps which must be included when developing data acquisition software that needs to access the data bus via DIGIO.

REFERENCES

[1] J. F. Harvey. *A Data Acquisition Parallel Bus For Wind Tunnels At ARL.* Flight Mechanics Technical Memorandum 412. Aeronautical Research Laboratory, DSTO Australia, August 1989.

[2] *DRV11 User's Manual.* Order number EK-ADV11-OP-002, Digital Equipment Corporation, Massachusetts USA, April 1977.

[3] S. C. Johnson. *Efficient Implementation of Real-Time Programs Under the VAX/VMS Operating System.* NASA Technical Memorandum 86354, 1985.

[4] *VAX/VMS System Generation Utility Reference Manual.* Order number AA-Z433A-TE, Digital Equipment Corporation, Massachusetts USA, April 1986.

[5] *MicroVMS Programmer's Manual.* Order number AI-Z212B-TE, Digital Equipment Corporation, Massachusetts USA, April 1986.

[6] *Writing a Device Driver for VAX/VMS.* Order number AA-Y511B-TE, Digital Equipment Corporation, Massachusetts USA, April 1986.

[7] *MicroVMS Programming Support Manual.* Order number AI-DC87B-TE, Digital Equipment Corporation, Massachusetts USA, April 1986.

[8] *VAX FORTRAN User Manual.* Order number AA-D035E-TE, Digital Equipment Corporation, Massachusetts USA, June 1988.

APPENDIX A

MODIFICATIONS TO THE DRV11 CIRCUIT BOARD

A design fault in the circuitry of the DRV11 circuit board prevents the interface card from functioning properly. This has been corrected by modifying the circuitry of the Integrated Circuit chip at the right hand top corner on the component side of the board, as shown in Figure A.1. The modifications, as shown in detail in Figure A.2 are:–

- The electrical connection between pins 8 and 9 is broken by cutting the circuit path between the two pins on the circuit side of the board.
- A jumper is wired between pins 6 and 9 to provide electrical connection between the two pins.

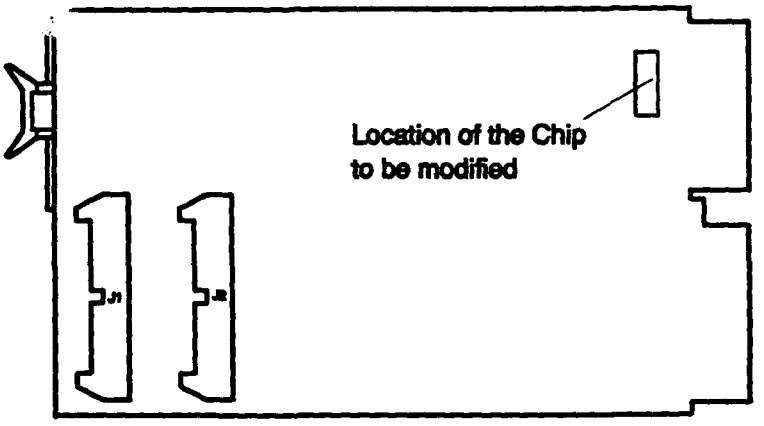

Figure A.1: Schematic layout of the DRV11 adapter board.

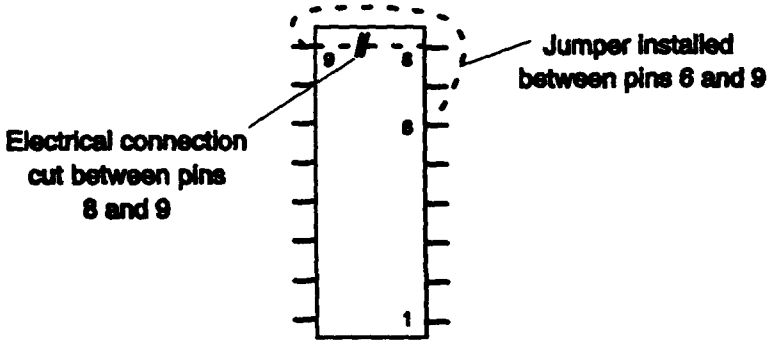

Figure A.2: Enlarged view of the chip to which modifications are made.

APPENDIX B

LISTING OF THE INCLUDE FILE DIGIO_INCLUDE.FOR

```
C ------------------------------------------------------------------------------
C************************** START OF DIGIO_INCLUDE.FOR ***************************
C ------------------------------------------------------------------------------

C -------------------------------
C       Define Module Usage Data
C -------------------------------

        LOGICAL         module_Table(0:15)      ! True if module electrically
                                                ! connected to the bus.
        CHARACTER*80    module_Name(0:15)       ! Returned when module
                                                ! interrogated.
C ----------------------------------
C       Define DIGIO Usage Indicators
C ----------------------------------

        INTEGER*2       num_Requestors          ! The number of processes
                                                ! connected to DIGIO. (max 5)
        LOGICAL         requestor_index_table(5)! True if process connected
                                                ! with this index.
        LOGICAL         data_usage_table(20),   ! True if the 50 location
                                                ! block of data_list is in use.
        1               dspon                   ! True if a display sub_process
                                                ! is executing.
C ----------------------------------
C       Define variable to data arrays
C ----------------------------------

        INTEGER*2       data_List(1000)         ! Array to hold bus data

C ---------------------------------------------
C       Define Global Common Block DIGIO_COMMON
C ---------------------------------------------

        COMMON /digio_common/   module_Table, module_Name,
        1                       num_Requestors, requestor_index_table,
        2                       requestor_start_add, data_usage_table,
        3                       data_List,
        5                       dspon

C ------------------------------------------------------------------------------
C ************************* END OF DIGIO_INCLUDE.FOR ******************************
C ------------------------------------------------------------------------------
```

APPENDIX C

LISTING OF THE INCLUDE FILE DIGIO_DEFS.FOR

```
C ------------------------------------------------------------------------------
C***************************** START OF DIGIO_DEFS.FOR ***************************
C ------------------------------------------------------------------------------

C ------------------------------------------------------------
C       Commonly used variables for communication with DIGIO
C ------------------------------------------------------------

        CHARACTER error_mbox_name*12,
        1         error_message*40,
        2         priority*1

        INTEGER*2 digio_mbox_chan,      ! digio mail box channel
        1         error_mbox_chan,      ! error mail box channel
        2         addr_index(50)        ! index of BPI bus addresses

        INTEGER*4 requestor_index,
        1         digio_status,
        2         data_start_addr,
        3         high_priority_efn,
        4         low_priority_efn,
        5         efn_success,
        6         efn_failure,
        7         efn_error,
        8         efn_timer,
        9         efn_mask

        PARAMETER ( low_priority_efn  = 65 , ! efn set by requesting process
        1           high_priority_efn = 64 ) ! depending on priority

        PARAMETER ( efn_timer = 95 )    ! efn of the timer to be set if
                                        ! DIGIO does not respond within
                                        ! the timeout period
        CHARACTER digio_mbox_name*(*)
        PARAMETER (digio_mbox_name = 'digio_mbox')

        CHARACTER digio_efn_cluster*(*)
        PARAMETER (digio_efn_cluster = 'dicluster')

C ------------------------------------------------------------------
C       The following common block is shared by routines within the
C       same process only.
C ------------------------------------------------------------------

        COMMON /process_common/ digio_mbox_chan, error_mbox_name,
        1                       error_mbox_chan, requestor_index,
        2                       data_start_addr, efn_success,
        3                       efn_failure, efn_error, efn_mask,
        4                       priority

C ------------------------------------------------------------------------------
C ***************************** END OF DIGIO_DEFS.FOR ****************************
C ------------------------------------------------------------------------------
```

APPENDIX D

LISTING OF THE SUBROUTINE DR_INIT.FOR

```
        SUBROUTINE dr_init
C ------------------------------------------------------------------------
C       This Subroutine initializes the DRV11 "driver". It does the
C               following:-
C
C       (1) Maps the page of physical memory containing the DR11's
C       registers (part of the Q22 bus or IO-page) to the processes
C       virtual memory. References to these locations on the Q22 bus
C       may then be made by reference to the array "dr_iopage" which
C       is 256 words (512 bytes - 1 page) long, contained in common
C       block "dr_common" which is linked to be page aligned via the
C       link option file "dr_init.opt"
C
C       (2) Connects to the DR11's interrupt vectors. Then whenever
C       interrupt A is set, event flag 10 will be raised, and whenever
C       interrupt B is set, the AST routine digio_error_ast will be
C       executed. This AST routine treats and signals all bus errors.
C
C ------------------------------------------------------------------------

C ----------------------
C       Include commons
C ----------------------

      INCLUDE '[datain.digio]dr_include.for/list'

C ---------------------------------
C       Define system routines used
C ---------------------------------

        EXTERNAL        io$_conintread

        INCLUDE         '($secdef)'

        INTEGER*4       status,                 ! system service routines.
        2               sys$assign,             ! assign a channel to mbox.
        7               sys$qio,                ! queue an i/o request.
        8               sys$crmpsc,             ! create a mapped section.
        6               lib$signal              ! signal error.

C ---------------------------------------
C       Define variables for CRMPSC Call
C ---------------------------------------

        INTEGER*4 dr_inadr(2),  ! Two longwords to contain io page address
        1         dr_retadr(2)  ! Two longwords to contain returned address

        INTEGER*4 pfn           ! io-page frame number
        PARAMETER ( pfn = '100007'X )

        INTEGER*4 mask          ! Longword to contain section characteristics
                                ! - pfn mapping, writeable

        PARAMETER ( mask = sec$m_pfnmap .OR. sec$m_wrt )

C ---------------------------------------
C       Define variables for ASSIGN Call
```

```
C ---------------------------------------

      INTEGER*2 dra_chan,     ! Channel number for interrupt A "device"
     1          drb_chan     ! Channel number for interrupt B "device"

C ---------------------------------------------------
C     Set up variables for Connect to Interrupt Call
C ---------------------------------------------------

      INTEGER*2 fcode         ! Function code word

      STRUCTURE /flag/
              UNION
               MAP
                INTEGER*2 switches /5/        ! cin$m_repeat or cin$m_efn
                INTEGER*2 efnum
               END MAP
               MAP
                INTEGER*4 flagword
               END MAP
              END UNION
      END STRUCTURE

      RECORD /flag/ dra_flags
      RECORD /flag/ drb_flags

      EXTERNAL digio_error_ast

C ------------------------------------------------------------------
C     ****************BEGINNING OF EXECUTABLE CODE*****************
C ------------------------------------------------------------------

C ---------------------------------------
C     Set up variables and map io-page
C ---------------------------------------

      dr_inadr(1) = %LOC( dr_iopage(0) )
      dr_inadr(2) = %LOC( dr_iopage(255) ) + 1

      status = SYS$CRMPSC( dr_inadr, dr_retadr, , %VAL( mask ), , , ,
     1                          , 1, %VAL( pfn ), , )

C ------------------------------------------------------------------
C     Check that the mapping was done correctly - input and return
C     start and end addresses should be the same
C ------------------------------------------------------------------

      IF( .NOT. status ) CALL LIB$SIGNAL( %VAL( status) )
      IF( ( dr_inadr(1) .NE. dr_retadr(1) ) .OR.
     1        ( dr_inadr(2) .NE. dr_retadr(2) ) ) THEN
              TYPE *, ' **** IO-Page not mapped correctly'
              TYPE 10, dr_inadr(1), dr_inadr(2)
10            FORMAT ( ' Input addresses are :- ', Z6, 5X, Z6)
              TYPE 20, dr_retadr(1), dr_retadr(2)
20            FORMAT ( ' Returned addresses are :- ', Z6, 5X, Z6)
              CALL LIB$SIGNAL( %VAL( status) )
      END IF

C ---------------------------------------------------
C     Assign both "devices" and get channel numbers
C ---------------------------------------------------
```

```
        status = SYS$ASSIGN( '_OAA0:', dra_chan, , )
        IF( .NOT. status ) CALL LIB$SIGNAL( %VAL( status) )
        status = SYS$ASSIGN( '_OAB0:', drb_chan, , )
        IF( .NOT. status ) CALL LIB$SIGNAL( %VAL( status) )

C -------------------------------------------
C       Connect to interrupts on both channels
C -------------------------------------------

C ------------------------------------------------------
C       For OAA0: - REQ A - the done flag - raise efn 1
C ------------------------------------------------------

        fcode = %LOC(io$_ConintRead)
        dra_flags.efnum = 1
        status = SYS$QIO ( , %VAL (dra_chan ), %VAL( fcode ), , , , , ,
        1       %VAL ( dra_flags.flagword ), , , )
        IF( .NOT. status ) CALL LIB$SIGNAL( %VAL( status) )

C ------------------------------------------------------------------
C       For OAB0: - REQ B - the error flag - go to digio_error_ast
C ------------------------------------------------------------------

        drb_flags.efnum = 0
        drb_flags.switches = 4
        status = SYS$QIO ( , %VAL (drb_chan ), %VAL( fcode ), , , , , ,
        1       %VAL ( drb_flags.flagword ), digio_error_ast, , %val(2) )
        IF( .NOT. status ) CALL LIB$SIGNAL( %VAL( status) )

C ---------------------------------
C       Complete - return to caller
C ---------------------------------

        RETURN
        END
```

APPENDIX E

LIST OF BUS ADDRESSES IN LOW SPEED WIND TUNNEL

```
C ------------------------------------------------------------------------------
C  ************************** START OF BUS_ADDRESSES **************************
C ------------------------------------------------------------------------------

C -------------------------------------------------------------
C       Definitions of Data Acquisition Parallel Bus addresses
C       allocated in LSWT  -  to be included in DIGIO.FOR
C -------------------------------------------------------------

        INTEGER*2       address_list (0:1599)

C ---------------------------------------------
C       Addresses that generate special events
C ---------------------------------------------

        DATA    address_list(0) / '0000'X /     ! NOP
        DATA    address_list(1) / 'FFFF'X /     ! MASTER RESET
        DATA    address_list(2) / '1'X /        ! Trigger all Modules

C -----------------------------------------------------
C       Addresses relating to the Tunnel Parameter Module
C -----------------------------------------------------

        DATA    address_list(500) / '8770'X /   ! V - data value read
        DATA    address_list(501) / '8772'X /   ! M - data value read
        DATA    address_list(502) / '8774'X /   ! Delta P - data value read
        DATA    address_list(503) / '8776'X /   ! Ps - data value read
        DATA    address_list(504) / '8778'X /   ! T - data value read
        DATA    address_list(505) / '877A'X /   ! Re - data value read
        DATA    address_list(506) / '877F'X /   ! V set write
        DATA    address_list(507) / '877D'X /   ! M set write
        DATA    address_list(508) / '8783'X /   ! Clear screen options write
        DATA    address_list(509) / '8789'X /   ! Raster code options write
        DATA    address_list(510) / '878F'X /   ! Large graphics options write
        DATA    address_list(511) / '8795'X /   ! Small graphics options write
        DATA    address_list(512) / '8779'X /   ! Air temperature display write
        DATA    address_list(513) / '87A7'X /   ! Fault/service message write
        DATA    address_list(514) / '87A1'X /   ! Display Message (write)
        DATA    address_list(515) / '87AD'X /   ! Write into master message buffer
        DATA    address_list(516) / '877B'X /   ! Load Ren No ref len (write)

C *** Display control commands
        DATA    address_list(520) / '8769'X/    ! Change display Mode
        DATA    address_list(521) / '8783'X/    ! Clear display
        DATA    address_list(522) / '8785'X/    ! Change background rastar
        DATA    address_list(523) / '8787'X/    ! Select large char display
        DATA    address_list(524) / '8789'X/    ! Select medium char display
        DATA    address_list(525) / '878B'X/    ! Select display on line 13
        DATA    address_list(526) / '878D'X/    ! Select display on line 14 & 15

C *** Send data commands
        DATA    address_list(530) / '8791'X/    ! Send message for display
        DATA    address_list(531) / '8793'X/    ! Send first label
        DATA    address_list(532) / '8795'X/    ! Send first number
        DATA    address_list(533) / '8797'X/    ! Send second label
        DATA    address_list(534) / '8799'X/    ! Send second number
        DATA    address_list(535) / '879B'X/    ! Send third label
```

```
      DATA    address_list(536) / '879D'X/    ! Send third number

      DATA    address_list(540) / '87F0'X/    ! Input Coef ID for update
      DATA    address_list(541) / '87F1'X/    ! Write Coef ID for update
      DATA    address_list(542) / '87F2'X/    ! Input Coef ID for read
      DATA    address_list(543) / '87F3'X/    ! Write Coef ID for read
      DATA    address_list(544) / '87AC'X/    ! Read Master Message buffer
      DATA    address_list(545) / '87AD'X/    ! Write Master Message buffer

C -------------------------------------------------
C       Addresses relating to the SG amplifier module
C -------------------------------------------------

C *** Data Value reads of all channels
      DATA    address_list(100) / '8169'X /   ! Trigger conversion on all ADC's
      DATA    address_list(101) / '81F4'X /   ! Read ADC conversion buffer
      DATA    address_list(102) / '8170'X /   ! Read ADC channel 1
      DATA    address_list(103) / '8172'X /   ! Read ADC channel 2
      DATA    address_list(104) / '8174'X /   ! Read ADC channel 3
      DATA    address_list(105) / '8176'X /   ! Read ADC channel 4
      DATA    address_list(106) / '8178'X /   ! Read ADC channel 5
      DATA    address_list(107) / '817A'X /   ! Read ADC channel 6

C *** Reads for channels in pairs
      DATA    address_list(108) / '8171'X /   ! Trigger Channels 1 & 2
      DATA    address_list(109) / '81D0'X /   ! Read ADC channel 1 status buffer
      DATA    address_list(110) / '8170'X /   ! Read Channel 1
      DATA    address_list(111) / '81D2'X /   ! Read ADC channel 2 status buffer
      DATA    address_list(112) / '8172'X /   ! Read Channel 2
      DATA    address_list(113) / '8175'X /   ! Trigger Channels 3 & 4
      DATA    address_list(114) / '81D4'X /   ! Read ADC channel 3 status buffer
      DATA    address_list(115) / '8174'X /   ! Read Channel 3
      DATA    address_list(116) / '81D6'X /   ! Read ADC channel 4 status buffer
      DATA    address_list(117) / '8176'X /   ! Read Channel 4
      DATA    address_list(118) / '8177'X /   ! Trigger Channels 5 & 6
      DATA    address_list(119) / '81D8'X /   ! Read ADC channel 5 status buffer
      DATA    address_list(120) / '8178'X /   ! Read Channel 5
      DATA    address_list(121) / '81DA'X /   ! Read ADC channel 6 status buffer
      DATA    address_list(122) / '817A'X /   ! Read Channel 6

C *** Calibration Relay Operations
      DATA    address_list(123) / '81F0'X /   ! Read calibration relay status
      DATA    address_list(124) / '81F1'X /   ! Turn Calibration Relay 'ON'
      DATA    address_list(125) / '81F1'X /   ! Turn Calibration Relay 'OFF'

C -------------------------------------------------
C       Addresses relating to the Inclinometer module
C -------------------------------------------------

      DATA    address_list(150) / '8A63'X /   ! Trigger conversion on all ADC
      DATA    address_list(151) / '8A70'X /   ! Read ADC 1, channel 1 - X
      DATA    address_list(152) / '8A72'X /   ! Read ADC 1, channel 2 - Y
      DATA    address_list(153) / '8A74'X /   ! Read ADC 2, channel 1 - Z
      DATA    address_list(154) / '8A76'X /   ! Read ADC 2, channel 2 - temp

      DATA    address_list(155) / '8A71'X /   ! Trigger ADC 1
      DATA    address_list(156) / '8A75'X /   ! Trigger ADC 2

      DATA    address_list(157) / '8A78'X /   ! Read Roll as calculated
      DATA    address_list(158) / '8A7A'X /   ! Read Pitch as calculated
      DATA    address_list(159) / '8A7C'X /   ! Read required Roll Offset
      DATA    address_list(160) / '8A7E'X /   ! Read required Pitch Offset
```

```
      DATA    address_list(161) / '8A7D'X /   ! Write Cal. Roll Offset
      DATA    address_list(162) / '8A7F'X /   ! Write Cal. Pitch Offset
      DATA    address_list(163) / '8AAB'X /   ! Model Alignment
      DATA    address_list(164) / '8AAD'X /   ! Select Qflex Transducer


C -----------------------------------------------
C       Addresses relating to the Actuator module
C -----------------------------------------------

      DATA    address_list(800) / '8B69'X /   ! Trigger simultaneous move
      DATA    address_list(801) / '8BD1'X /   ! Manual drive - channel
      DATA    address_list(802) / '8BD3'X /   ! Manual drive - move
      DATA    address_list(803) / '8BD5'X /   ! Manual drive - exit
      DATA    address_list(804) / '8B66'X /   ! Read calibration coeff.
      DATA    address_list(805) / '8B60'X /   ! Read error code
      DATA    address_list(806) / '8B61'X /   ! Reset error status
      DATA    address_list(807) / '8B65'X /   ! Clear status/error buffer
      DATA    address_list(808) / '8B67'X /   ! Clear error & flag bits
      DATA    address_list(809) / '8BD0'X /   ! Read Motor Status Register

C *** Channel 1
      DATA    address_list(810) / '8B70'X /   ! LVDT Reading to master
      DATA    address_list(811) / '8B71'X /   ! Angle to Move to
      DATA    address_list(812) / '8B72'X /   ! Current angle
      DATA    address_list(813) / '8B73'X /   ! Set upper limit
      DATA    address_list(814) / '8B74'X /   ! Read upper limit
      DATA    address_list(815) / '8B75'X /   ! Set lower limit
      DATA    address_list(816) / '8B76'X /   ! Read lower limit
      DATA    address_list(817) / '8B77'X /   ! Set Offset angle

C *** Channel 2
      DATA    address_list(820) / '8B78'X /   ! LVDT Reading to master
      DATA    address_list(821) / '8B79'X /   ! Angle to Move to
      DATA    address_list(822) / '8B7A'X /   ! Current angle
      DATA    address_list(823) / '8B7B'X /   ! Set upper limit
      DATA    address_list(824) / '8B7C'X /   ! Read upper limit
      DATA    address_list(825) / '8B7D'X /   ! Set lower limit
      DATA    address_list(826) / '8B7E'X /   ! Read lower limit
      DATA    address_list(827) / '8B7F'X /   ! Set Offset angle

C *** Channel 3
      DATA    address_list(830) / '8B80'X /   ! LVDT Reading to master
      DATA    address_list(831) / '8B81'X /   ! Angle to Move to
      DATA    address_list(832) / '8B82'X /   ! Current angle
      DATA    address_list(833) / '8B83'X /   ! Set upper limit
      DATA    address_list(834) / '8B84'X /   ! Read upper limit
      DATA    address_list(835) / '8B85'X /   ! Set lower limit
      DATA    address_list(836) / '8B86'X /   ! Read lower limit
      DATA    address_list(837) / '8B87'X /   ! Set Offset angle

C *** Channel 4
      DATA    address_list(840) / '8B88'X /   ! LVDT Reading to master
      DATA    address_list(841) / '8B89'X /   ! Angle to Move to
      DATA    address_list(842) / '8B8A'X /   ! Current angle
      DATA    address_list(843) / '8B8B'X /   ! Set upper limit
      DATA    address_list(844) / '8B8C'X /   ! Read upper limit
      DATA    address_list(845) / '8B8D'X /   ! Set lower limit
      DATA    address_list(846) / '8B8E'X /   ! Read lower limit
      DATA    address_list(847) / '8B8F'X /   ! Set Offset angle

C *** Channel 5
```

```
      DATA    address_list(850) / '8B90'X /   ! LVDT Reading to master
      DATA    address_list(851) / '8B91'X /   ! Angle to Move to
      DATA    address_list(852) / '8B92'X /   ! Current angle
      DATA    address_list(853) / '8B93'X /   ! Set upper limit
      DATA    address_list(854) / '8B94'X /   ! Read upper limit
      DATA    address_list(855) / '8B95'X /   ! Set lower limit
      DATA    address_list(856) / '8B96'X /   ! Read lower limit
      DATA    address_list(857) / '8B97'X /   ! Set Offset angle

C *** Channel 6
      DATA    address_list(860) / '8B98'X /   ! LVDT Reading to master
      DATA    address_list(861) / '8B99'X /   ! Angle to Move to
      DATA    address_list(862) / '8B9A'X /   ! Current angle
      DATA    address_list(863) / '8B9B'X /   ! Set upper limit
      DATA    address_list(864) / '8B9C'X /   ! Read upper limit
      DATA    address_list(865) / '8B9D'X /   ! Set lower limit
      DATA    address_list(866) / '8B9E'X /   ! Read lower limit
      DATA    address_list(867) / '8B9F'X /   ! Set Offset angle

C *** Channel 7
      DATA    address_list(870) / '8BA0'X /   ! LVDT Reading to master
      DATA    address_list(871) / '8BA1'X /   ! Angle to Move to
      DATA    address_list(872) / '8BA2'X /   ! Current angle
      DATA    address_list(873) / '8BA3'X /   ! Set upper limit
      DATA    address_list(874) / '8BA4'X /   ! Read upper limit
      DATA    address_list(875) / '8BA5'X /   ! Set lower limit
      DATA    address_list(876) / '8BA6'X /   ! Read lower limit
      DATA    address_list(877) / '8BA7'X /   ! Set Offset angle

C *** Channel 8
      DATA    address_list(880) / '8BA8'X /   ! LVDT Reading to master
      DATA    address_list(881) / '8BA9'X /   ! Angle to Move to
      DATA    address_list(882) / '8BAA'X /   ! Current angle
      DATA    address_list(883) / '8BAB'X /   ! Set upper limit
      DATA    address_list(884) / '8BAC'X /   ! Read upper limit
      DATA    address_list(885) / '8BAD'X /   ! Set lower limit
      DATA    address_list(886) / '8BAE'X /   ! Read lower limit
      DATA    address_list(887) / '8BAF'X /   ! Set Offset angle

C *** Channel 9
      DATA    address_list(890) / '8BB0'X /   ! LVDT Reading to master
      DATA    address_list(891) / '8BB1'X /   ! Angle to Move to
      DATA    address_list(892) / '8BB2'X /   ! Current angle
      DATA    address_list(893) / '8BB3'X /   ! Set upper limit
      DATA    address_list(894) / '8BB4'X /   ! Read upper limit
      DATA    address_list(895) / '8BB5'X /   ! Set lower limit
      DATA    address_list(896) / '8BB6'X /   ! Read lower limit
      DATA    address_list(897) / '8BB7'X /   ! Set Offset angle
```

```
C *** Channel 10
        DATA    address_list(900) / '8BB8'X /   ! LVDT Reading to master
        DATA    address_list(901) / '8BB9'X /   ! Angle to Move to
        DATA    address_list(902) / '8BBA'X /   ! Current angle
        DATA    address_list(903) / '8BBB'X /   ! Set upper limit
        DATA    address_list(904) / '8BBC'X /   ! Read upper limit
        DATA    address_list(905) / '8BBD'X /   ! Set lower limit
        DATA    address_list(906) / '8BBE'X /   ! Read lower limit
        DATA    address_list(907) / '8BBF'X /   ! Set Offset angle

C *** Channel 11
        DATA    address_list(910) / '8BC0'X /   ! LVDT Reading to master
        DATA    address_list(911) / '8BC1'X /   ! Angle to Move to
        DATA    address_list(912) / '8BC2'X /   ! Current angle
        DATA    address_list(913) / '8BC3'X /   ! Set upper limit
        DATA    address_list(914) / '8BC4'X /   ! Read upper limit
        DATA    address_list(915) / '8BC5'X /   ! Set lower limit
        DATA    address_list(916) / '8BC6'X /   ! Read lower limit
        DATA    address_list(917) / '8BC7'X /   ! Set Offset angle

C *** Channel 12
        DATA    address_list(920) / '8BC8'X /   ! LVDT Reading to master
        DATA    address_list(921) / '8BC9'X /   ! Angle to Move to
        DATA    address_list(922) / '8BCA'X /   ! Current angle
        DATA    address_list(923) / '8BCB'X /   ! Set upper limit
        DATA    address_list(924) / '8BCC'X /   ! Read upper limit
        DATA    address_list(925) / '8BCD'X /   ! Set lower limit
        DATA    address_list(926) / '8BCE'X /   ! Read lower limit
        DATA    address_list(927) / '8BCF'X /   ! Set Offset angle

C *** Actuator Calibration Coefficients
        DATA    address_list(930) / '8BDB'X /  ! Turn Manual Pulse Mode ON
        DATA    address_list(931) / '8BDD'X /  ! Turn Manual Pulse Mode OFF

C ------------------------------------------------------------------------------
C  ***************************  END OF BUS_ADDRESSES  ***************************
C ------------------------------------------------------------------------------
```

APPENDIX F

LIST OF BUS ADDRESSES IN VARIABLE PRESSURE TRANSONIC WIND TUNNEL

```
C ------------------------------------------------------------------------------
C  ************************** START OF BUS_ADDRESSES **************************
C ------------------------------------------------------------------------------

C ------------------------------------------------------------
C       Definitions of Data Acquisition Parallel Bus addresses
C           -  to be included in DIGID.FOR
C ------------------------------------------------------------

        INTEGER*2       address_list (0:1599)

C ---------------------------------------------
C       Addresses that generate special events
C ---------------------------------------------

        DATA    address_list(0) / '0000'X /     ! NOP
        DATA    address_list(1) / 'FFFF'X /     ! MASTER RESET
        DATA    address_list(2) / '1'X /        ! Trigger all Modules

C ------------------------------------------------------
C       Addresses relating to the Tunnel Parameter Module
C ------------------------------------------------------

        DATA    address_list(500) / '8570'X /   ! V - data value read
        DATA    address_list(501) / '8572'X /   ! M - data value read
        DATA    address_list(502) / '8574'X /   ! H - data value read
        DATA    address_list(503) / '8576'X /   ! P - data value read
        DATA    address_list(504) / '8578'X /   ! T - data value read
        DATA    address_list(505) / '857A'X /   ! Re - data value read
        DATA    address_list(506) / '8571'X /   ! V set write
        DATA    address_list(507) / '8573'X /   ! M set write
        DATA    address_list(508) / '8583'X /   ! Clear screen options write
        DATA    address_list(509) / '8585'X /   ! Raster code options write
        DATA    address_list(510) / '8587'X /   ! Large graphics options write
        DATA    address_list(511) / '8589'X /   ! Small graphics options write
        DATA    address_list(512) / '858B'X /   ! Air temperature display write
        DATA    address_list(513) / '8595'X /   ! Fault/service message write
        DATA    address_list(514) / '858D'X /   ! Display Message (write)
        DATA    address_list(515) / '8591'X /   ! Write into master message buffer
        DATA    address_list(516) / '8575'X /   ! Load Ren No ref len (write)

C *** Display control commands
        DATA    address_list(520) / '8569'X/    ! Change display Mode
        DATA    address_list(521) / '8583'X/    ! Clear display
        DATA    address_list(522) / '8585'X/    ! Change background raster
        DATA    address_list(523) / '8587'X/    ! Select large char display
        DATA    address_list(524) / '8589'X/    ! Select medium char display
        DATA    address_list(525) / '858B'X/    ! Select display on line 13
        DATA    address_list(526) / '858D'X/    ! Select display on line 14 & 15

C *** Send data commands
        DATA    address_list(530) / '8591'X/    ! Send message for display
        DATA    address_list(531) / '8593'X/    ! Send first label
        DATA    address_list(532) / '8595'X/    ! Send first number
        DATA    address_list(533) / '8597'X/    ! Send second label
        DATA    address_list(534) / '8599'X/    ! Send second number
```

```
      DATA    address_list(535) / '859B'X/    ! Send third label
      DATA    address_list(536) / '859D'X/    ! Send third number

      DATA    address_list(540) / '856C'X/    ! Input Coeff ID for update
      DATA    address_list(541) / '856D'X/    ! Write Coeff ID for update
      DATA    address_list(542) / '856E'X/    ! Input Coeff ID for read
      DATA    address_list(543) / '856F'X/    ! Write Coeff ID for read
      DATA    address_list(544) / '8590'X/    ! Read Master Message Buffer

C -------------------------------------------------------
C       Addresses relating to the SG amplifier module
C -------------------------------------------------------

C *** Data Value reads of all channels
      DATA    address_list(100) / '8169'X /   ! Trigger conversion on all ADC's
      DATA    address_list(101) / '81F4'X /   ! Read ADC conversion buffer
      DATA    address_list(102) / '8170'X /   ! Read ADC channel 1
      DATA    address_list(103) / '8172'X /   ! Read ADC channel 2
      DATA    address_list(104) / '8174'X /   ! Read ADC channel 3
      DATA    address_list(105) / '8176'X /   ! Read ADC channel 4
      DATA    address_list(106) / '8178'X /   ! Read ADC channel 5
      DATA    address_list(107) / '817A'X /   ! Read ADC channel 6

C *** Reads for channels in pairs
      DATA    address_list(108) / '8171'X /   ! Trigger Channels 1 & 2
      DATA    address_list(109) / '81D0'X /   ! Read ADC channel 1 status buffer
      DATA    address_list(110) / '8170'X /   ! Read Channel 1
      DATA    address_list(111) / '81D2'X /   ! Read ADC channel 2 status buffer
      DATA    address_list(112) / '8172'X /   ! Read Channel 2
      DATA    address_list(113) / '8175'X /   ! Trigger Channels 3 & 4
      DATA    address_list(114) / '81D4'X /   ! Read ADC channel 3 status buffer
      DATA    address_list(115) / '8174'X /   ! Read Channel 3
      DATA    address_list(116) / '81D6'X /   ! Read ADC channel 4 status buffer
      DATA    address_list(117) / '8176'X /   ! Read Channel 4
      DATA    address_list(118) / '8177'X /   ! Trigger Channels 5 & 6
      DATA    address_list(119) / '81D8'X /   ! Read ADC channel 5 status buffer
      DATA    address_list(120) / '8178'X /   ! Read Channel 5
      DATA    address_list(121) / '81DA'X /   ! Read ADC channel 6 status buffer
      DATA    address_list(122) / '817A'X /   ! Read Channel 6

C *** Calibration Relay Operations
      DATA    address_list(123) / '81F0'X /   ! Read calibration relay status
      DATA    address_list(124) / '81F1'X /   ! Turn Calibration Relay 'ON'
      DATA    address_list(125) / '81F1'X /   ! Turn Calibration Relay 'OFF'
```

```
C ------------------------------------------------
C       Addresses relating to the Scani-valve module
C ------------------------------------------------

      DATA    address_list(300) / '8361'X /   ! Trigger data acquisition
      DATA    address_list(301) / '8364'X /   ! Read status of buffer
      DATA    address_list(302) / '8365'X /   ! Clear all buffers
      DATA    address_list(303) / '836C'X /   ! Read selected scani-valve
      DATA    address_list(304) / '8369'X /   ! Set power settings
      DATA    address_list(305) / '836B'X /   ! Set operation mode code

C *** Calibration Relay Operations
      DATA    address_list(308) / '83F0'X / ! Read calibration relay status
      DATA    address_list(309) / '83F1'X / ! Toggle Calibration Relay ON/OFF

C *** Trigger ADC read on scani-valve
      DATA    address_list(311) / '8371'X /   ! Read Scani-valve 1
      DATA    address_list(312) / '8373'X /   ! Read Scani-valve 2
      DATA    address_list(313) / '8375'X /   ! Read Scani-valve 3
      DATA    address_list(314) / '8377'X /   ! Read Scani-valve 4
      DATA    address_list(315) / '8379'X /   ! Read Scani-valve 5
      DATA    address_list(316) / '837B'X /   ! Read Scani-valve 6

C *** Read ADC cards of all channels
      DATA    address_list(321) / '8370'X /   ! Read Scani-valve 1
      DATA    address_list(322) / '8372'X /   ! Read Scani-valve 2
      DATA    address_list(323) / '8374'X /   ! Read Scani-valve 3
      DATA    address_list(324) / '8376'X /   ! Read Scani-valve 4
      DATA    address_list(325) / '8378'X /   ! Read Scani-valve 5
      DATA    address_list(326) / '837A'X /   ! Read Scani-valve 6

C *** Set number of selected ports on scanivalve
      DATA    address_list(331) / '8391'X /   ! Scani-valve 1
      DATA    address_list(332) / '8393'X /   ! Scani-valve 2
      DATA    address_list(333) / '8395'X /   ! Scani-valve 3
      DATA    address_list(334) / '8397'X /   ! Scani-valve 4
      DATA    address_list(335) / '8399'X /   ! Scani-valve 5
      DATA    address_list(336) / '839B'X /   ! Scani-valve 6

C *** Read number of selected ports on scanivalve
      DATA    address_list(341) / '8390'X /   ! Scani-valve 1
      DATA    address_list(342) / '8392'X /   ! Scani-valve 2
      DATA    address_list(343) / '8394'X /   ! Scani-valve 3
      DATA    address_list(344) / '8396'X /   ! Scani-valve 4
      DATA    address_list(345) / '8398'X /   ! Scani-valve 5
      DATA    address_list(346) / '839A'X /   ! Scani-valve 6

C *** Set the currently selected port on scanivalve
      DATA    address_list(351) / '839D'X /   ! Scani-valve 1
      DATA    address_list(352) / '839F'X /   ! Scani-valve 2
      DATA    address_list(353) / '83A1'X /   ! Scani-valve 3
      DATA    address_list(354) / '83A3'X /   ! Scani-valve 4
      DATA    address_list(355) / '83A5'X /   ! Scani-valve 5
      DATA    address_list(356) / '83A7'X /   ! Scani-valve 6

C *** Read the currently selected port on scanivalve
      DATA    address_list(361) / '839C'X /   ! Scani-valve 1
      DATA    address_list(362) / '839E'X /   ! Scani-valve 2
      DATA    address_list(363) / '83A0'X /   ! Scani-valve 3
      DATA    address_list(364) / '83A2'X /   ! Scani-valve 4
      DATA    address_list(365) / '83A4'X /   ! Scani-valve 5
      DATA    address_list(366) / '83A6'X /   ! Scani-valve 6
```

```
C *** Set the settling time used on scanivalve (in ms)
      DATA    address_list(371) / '83A9'I /   ! Scani-valve 1
      DATA    address_list(372) / '83AB'I /   ! Scani-valve 2
      DATA    address_list(373) / '83AD'I /   ! Scani-valve 3
      DATA    address_list(374) / '83AF'I /   ! Scani-valve 4
      DATA    address_list(375) / '83B1'I /   ! Scani-valve 5
      DATA    address_list(376) / '83B3'I /   ! Scani-valve 6

C *** Read the settling time used on scanivalve
      DATA    address_list(381) / '83A8'I /   ! Scani-valve 1
      DATA    address_list(382) / '83AA'I /   ! Scani-valve 2
      DATA    address_list(383) / '83AC'I /   ! Scani-valve 3
      DATA    address_list(384) / '83AE'I /   ! Scani-valve 4
      DATA    address_list(385) / '83B0'I /   ! Scani-valve 5
      DATA    address_list(386) / '83B2'I /   ! Scani-valve 6

C -------------------------------------------------------------------------------
C  ***************************  END OF BUS_ADDRESSES  ***************************
C -------------------------------------------------------------------------------
```

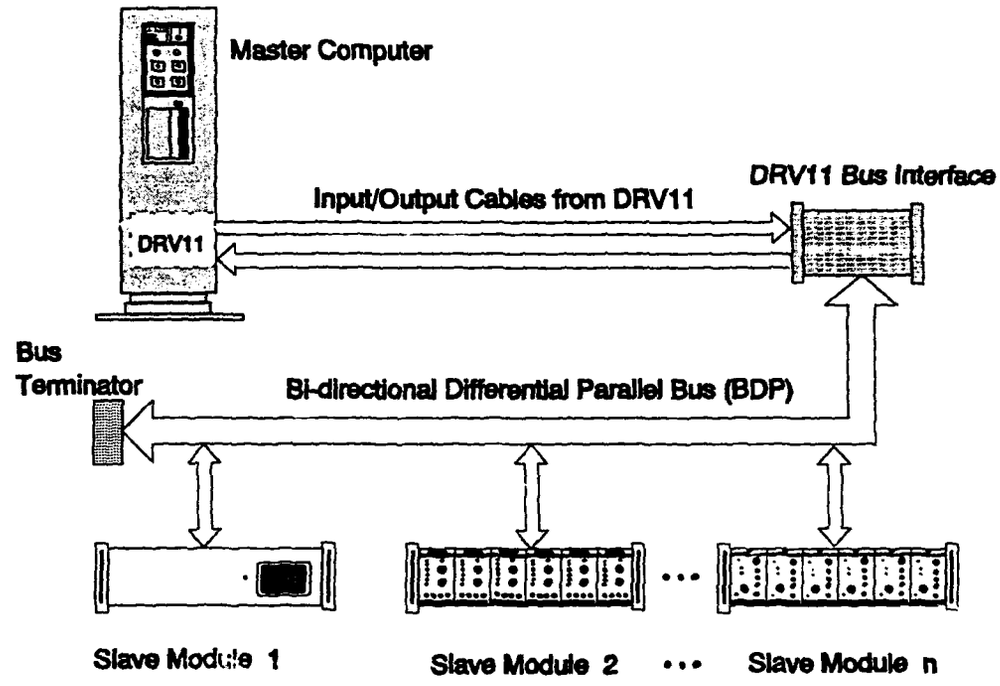

Figure 1: Configuration of the Wind Tunnel Data Acquisition System.

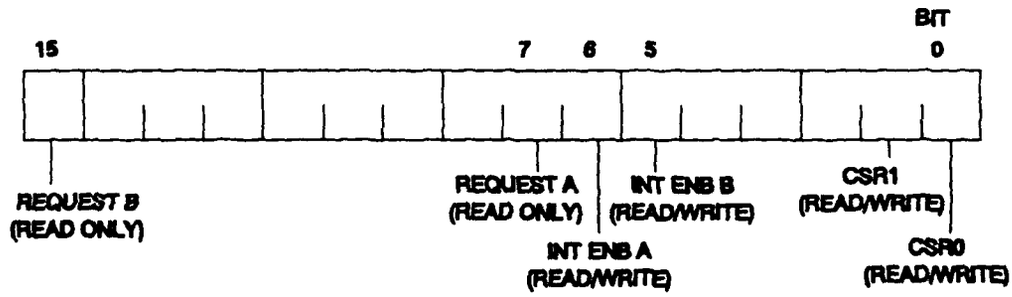

Figure 2: The word format of DRV11 Control Status Register.

DISTRIBUTION

AUSTRALIA

Department of Defence

Defence Central

Chief Defence Scientist
AS Science Corporate Management } shared copy
FAS Science Policy
Director Departmental Publications
Counsellor Defence Science, London (Doc Data sheet only)
Counsellor Defence Science, Washington (Doc Data sheet only)
Scientific Adviser Defence Central
OIC TRS Defence Central Library
Document Exchange Centre, DSTIC (8 copies)
Defence Intelligence Organisation
Librarian Defence Signals Directorate, (Doc Data sheet only)

Aeronautical Research Laboratory

Director
Library
Chief Air Operations Division
Authors: B.D. Fairley
S.S.W. Lam
N. Matheson (3 copies)
M. Glaister
L. MacLaren
Y. Link
P. Malone (3 copies)

Materials Research Laboratory

Director/Library

Defence Science & Technology Organisation - Salisbury

Library

Navy Office

Navy Scientific Adviser (3 copies Doc Data sheet only)
Naval Support Command
Superintendent, Naval Aircraft Logistics
Directorate of Aviation Projects - Navy

Army Office

Scientific Adviser - Army (Doc Data sheet only)
Engineering Development Establishment Library
US Army Research, Development and Standardisation Group (3 copies)

Air Force Office
Air Force Scientific Adviser (Doc Data sheet only)
Aircraft Research and Development Unit
Scientific Flight Group
Library
OIC ATF, ATS, RAAFSTT, WAGGA (2 copies)

HQ ADF
Director General Force Development (Air)

Department of Transport & Communication
Library

Statutory and State Authorities and Industry
Aero-Space Technologies Australia, Systems Division Librarian
ASTA Engineering, Document Control Office
Civil Aviation Authority
Hawker de Havilland Aust Pty Ltd, Victoria, Library
Hawker de Havilland Aust Pty Ltd, Bankstown, Library
Australian Nuclear Science and Technology Organisation
Gas & Fuel Corporation of Vic., Manager Scientific Services
SEC of Vic., Herman Research Laboratory, Library

Universities and Colleges
Adelaide
Barr Smith Library

Flinders
Library

LaTrobe
Library

Melbourne
Engineering Library

Monash
Hargrave Library

Newcastle
Library

New England
Library

Sydney
Engineering Library

NSW
Library, Australian Defence Force Academy

Queensland
Library

Tasmania
Engineering Library

Western Australia
Library

RMIT
Library

University College of the Northern Territory
Library

INDIA
CAARC Coordinator Aerodynamics

UNITED KINGDOM
CAARC Coordinator Aerodynamics

UNITED STATES OF AMERICA
NASA Scientific and Technical Information Facility

SPARES (4 COPIES)

TOTAL (70 COPIES)

AL 149 DEPARTMENT OF DEFENCE

DOCUMENT CONTROL DATA

PAGE CLASSIFICATION
UNCLASSIFIED

PRIVACY MARKING

1a. AR NUMBER	1b. ESTABLISHMENT NUMBER	2. DOCUMENT DATE	3. TASK NUMBER
AR-007-135	ARL-TR-14	MARCH 1993	DST 92/459

4. TITLE

A SOFTWARE INTERFACE FOR THE ARL WIND TUNNEL DATA ACQUISITION SYSTEM

5. SECURITY CLASSIFICATION (PLACE APPROPRIATE CLASSIFICATION IN BOX(S) IE. SECRET (S), CONF. (C) RESTRICTED (R), UNCLASSIFIED (U)).

DOCUMENT	TITLE	ABSTRACT
U	U	U

6. NO. PAGES
48

7. NO. REFS.
8

8. AUTHOR(S)

B.D. FAIRLIE
S.S.W. LAM

9. DOWNGRADING/DELIMITING INSTRUCTIONS

Not applicable.

10. CORPORATE AUTHOR AND ADDRESS

AERONAUTICAL RESEARCH LABORATORY
AIR OPERATIONS DIVISION
506 LORIMER STREET
FISHERMENS BEND VIC 3207

11. OFFICE/POSITION RESPONSIBLE FOR:

SPONSOR DSTO
SECURITY
DOWNGRADING
APPROVAL CAOD

12. SECONDARY DISTRIBUTION (OF THIS DOCUMENT)

Approved for public release.
OVERSEAS ENQUIRIES OUTSIDE STATED LIMITATIONS SHOULD BE REFERRED THROUGH DSTIC, ADMINISTRATIVE SERVICES BRANCH, DEPARTMENT OF DEFENCE, ANZAC PARK WEST OFFICES, ACT 2601

13a. THIS DOCUMENT MAY BE ANNOUNCED IN CATALOGUES AND AWARENESS SERVICES AVAILABLE TO ....

No limitations.

13b. CITATION FOR OTHER PURPOSES (IE. CASUAL ANNOUNCEMENT) MAY BE [X] UNRESTRICTED OR [ ] AS FOR 13a.

14. DESCRIPTORS

Transonic wind tunnels
Low speed wind tunnels
Aeronautical Research Laboratory
Data acquisition
Software tool support interfaces
Computer programs

15. DISCAT SUBJECT CATEGORIES
010101

16 ABSTRACT

A software interface for the data acquisition system has been developed on a MicroVAX II computer for the Transonic and Low Speed wind tunnels at ARL. The software is responsible for handling instrumentation control and data transfer requests between the data acquisition software and the parallel data bus via a DRV11 parallel I/O interface adapter. Access to the DRV11 registers is effected by direct mapping of the Q22-Bus I/O page to program variables, giving fast and efficient transfer of data to and from the parallel data bus. Up to five processes may access the parallel data bus at one time via this software interface thus allowing great flexibility in the development of data acquisition software. This report details the necessary programming steps which must be included in data acquisition software to access the parallel data bus via the software interface.

PAGE CLASSIFICATION
UNCLASSIFIED
PRIVACY MARKING

THIS PAGE IS TO BE USED TO RECORD INFORMATION WHICH IS REQUIRED BY THE ESTABLISHMENT FOR ITS OWN USE BUT WHICH WILL NOT BE ADDED TO THE DISTIS DATA UNLESS SPECIFICALLY REQUESTED.

16. ABSTRACT (CONT).

17. IMPRINT

AERONAUTICAL RESEARCH LABORATORY, MELBOURNE

18. DOCUMENT SERIES AND NUMBER	19. WA NUMBER	20. TYPE OF REPORT AND PERIOD COVERED
Technical Report 14	54 527F	

21. COMPUTER PROGRAMS USED

22. ESTABLISHMENT FILE REF.(S)

23. ADDITIONAL INFORMATION (AS REQUIRED)